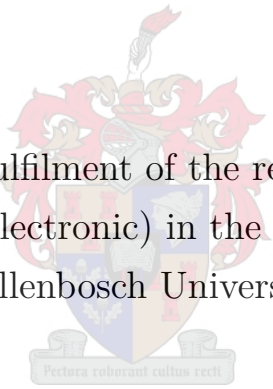
Cooperative Navigation for Multiple Autonomous Ground Vehicles (AGVs) with Kinematic Constraints

R. M. Viljoen



Thesis presented in partial fulfilment of the requirements for the degree of Master of Engineering (Electronic) in the Faculty of Engineering at Stellenbosch University.

Supervisors: Dr J. A. A. Engelbrecht, Prof. H. A. Engelbrecht
Department of Electrical and Electronic Engineering

December 2020

Acknowledgements

I would like to thank the following:

- Dr J. A. A. Engelbrecht and Prof. H. A. Engelbrecht for their guidance and assistance throught the course of this project.
- The Alphawave Group for the bursary and financial support that they gave me during my postgraduate studies.
- My family, for their support and aid during my studies. I am especially grateful for their unwavering support during the covid-19 lockdown.
- For my friends at the Electronic Systems Laboratory, who provided me with encouragement and interesting conversations throughout my studies.
- The technical staff at the E&E Department, including Wessel Croukamp and Wynand van Eeden.

UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Plagiaatverklaring / *Plagiarism Declaration*

1. Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.
   Plagiarism is the use of ideas, material and other intellectual property of another's work and to present is as my own.

2. Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.
   I agree that plagiarism is a punishable offence because it constitutes theft.

3. Ek verstaan ook dat direkte vertalings plagiaat is.
   I also understand that direct translations are plagiarism.

4. Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelikse aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.
   Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism

5. Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.
   I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.

Abstract

English

This thesis presents the development of a system for the cooperative navigation of several Autonomous Ground Vehicles (AGVs) within the same environment. A high-level system architecture is designed that includes the following modular components: a cooperative trajectory planner, a trajectory tracker, and a velocity controller. The cooperative trajectory planner forms the highest level subsystem, and is responsible for finding collision-free trajectories for each vehicle. It does this using a decentralised coordination strategy, allowing for a more distributive and resilient system. The planning is accomplished for each vehicle through the use of the Windowed Hierarchical Cooperative A* (WHCA*) multi-agent planning algorithm, modified so as to adhere to the kinematic constraints of the vehicles. The second subsystem is the trajectory tracking module, which uses a Model Predictive Control (MPC) strategy to control the vehicles to track the planned trajectories, while also taking the kinematic constraints of the vehicle into account.

Each of the subsystems were developed and tested using a simulation environment made with the ROS and Gazebo toolchain. This simulation environment was also used to test the overall performance of the integrated system. These tests were repeated using a practical setup with physical vehicles, so as to evaluate the performance of the system in a real world environment. In order to perform the practical tests, both the physical vehicles and a vehicle pose estimation system were designed and built. The purpose of the vehicle pose estimation system was to find and track the pose of the vehicles, which was required by both the trajectory planning and tracking algorithms. The vehicle pose estimation was accomplished through the use of the ArUco fiducial marker detection computer vision algorithm.

Both the simulation and practical tests show that the cooperative navigation algorithms were able to successfully plan and execute trajectories using a decentralised coordination strategy, resulting in collision free navigation for all the vehicles involved. Both the trajectory planning and the trajectory optimisation were able to execute within their allowed time frame, which means the cooperative navigation system is viable for real-time operation.

Afrikaans

Hierdie tesis beskryf die ontwerp van ’n stelsel wat gebruik kan word vir die gedesentraliseerde samewerkingsnavigasie van verskeie outonome grondvoertuie binne dieselfde omgewing. ’n Stelselargitektuur op hoë vlak is ontwerp wat die volgende modulêre komponente bevat: ’n koöperatiewe trajekbeplanner, ’n trajekuitvoerder, en ’n snelheidskontroleerder. Die koöperatiewe trajekbeplanner vorm die hoogste stelsel en is verantwoordelik vir die vind van botsingsvrye trajekte vir elke voertuig. Dit word gedoen met behulp van ’n gedesentraliseerde koördineringstrategie, wat ’n meer verspreidende en betroubare stelsel moontlik maak. Die beplanning word vir elke voertuig gedoen deur gebruik te maak van die Windowed Hierarchical Cooperative A* (WHCA*) veelagent beplanningsalgoritme, aangepas om te voldoen aan die kinematiese beperkings van die voertuie. Die tweede substelsel is die trajekuitvoeringmodule, wat gebruik maak van ’n Model Predictive Control (MPC) strategie om die betroubare uitvoering van die beplande trajekte te verseker, terwyl die kinematiese beperkings van die voertuig ook in ag geneem word.

Elk van die substelsels is ontwikkel en getoets met behulp van ’n simulasie-omgewing wat gemaak is met die ROS en Gazebo gereedskapsketting. Hierdie simulasie-omgewing is ook gebruik om die algehele optrede te toets sodra al die substelsels in ’n holistiese oplossing geïntegreer is. Hierdie toetse is herhaal met behulp van ’n praktiese opstelling met fisiese voertuie om die optrede van die stelsel in ’n werklike wêreldomgewing te evalueer. Om die praktiese toetse uit te voer, moes beide die fisiese voertuie en ’n voertuigopsporingstelsel ontwerp en gebou word. Die doel van die voertuigopsporingstelsel was om die geskatte posisie van die voertuig te verskaf, wat deur die trajekbeplanning en trajekuitvoering algoritmes vereis word. Die voertuigopsporing is gedoen deur die ArUco merker rekenaarvisie-algoritme te gebruik.

Beide die simulasie en praktiese toetse toon dat die koöperatiewe navigasie-algoritmes in staat was om trajekte suksesvol te beplan en uit te voer met behulp van die gedesentraliseerde koördineringstrategie, wat gelei het tot botsingsvrye navigasie vir al die betrokke voertuie. Beide die trajekbeplanning en die trajekoptimalisering kon binne hul toegelate tydsbestek uitgevoer word, wat beteken dat die koöperatiewe navigasiestelsel gebruikbaar is vir intydse werking.

Contents

List of Figures

List of Tables

Nomenclature

Variables and functions

$f(n)$	Cost of the shortest path from the start node s to the goal node t.
$g(n)$	Cost of the shortest path from the start node s to the current node n.
$h(n)$	Cost of the shortest path from the current node n to the goal node t.
$\hat{f}(n)$	Estimate of $f(n)$.
$\hat{g}(n)$	Estimate of $g(n)$.
$\hat{h}(n)$	Estimate of $h(n)$.
H	The heuristic lookup table generated using the reverse A* search.
Γ	Successor operator used during A* node expansion.
$\bar{x}$	Vector representing the state of the vehicle.
$\hat{x}$	Vector representing the estimated state of the vehicle.
x^{ref}	Reference state of vehicle during the trajectory optimisation process.
$f(\bar{x}, p)$	Objective function used for the trajectory optimisation process.
$g(\bar{x})$	Constraints function that is applied to the state vector during the optimisation process.
v	Linear velocity of the vehicle.
ω	Angular velocity of the vehicle.
L	Perpendicular distance between the two wheels of the vehicle.
r	Radius of the vehicle's wheel.
N_{nodes}	The total number of nodes that are needed to represent the vehicle's environment during the A* search.
f_{steps}	The frequency at which the stepper motor is driven.
T	The rotation and translation transform matrix
R	The rotational transform matrix.
α	The weighting used for the predicted state of the vehicle when performing the vehicle state estimation process.
β	The weighting used for the measured state of the vehicle when performing the vehicle state estimation process.

Acronyms and abbreviations

ADA*	Anytime Dynamic A*
AI	Artificial Intelligence
AGV	Autonomous Ground Vehicle
ARA*	Anytime Repairable A*
AVO	Acceleration-Velocity Obstacle
BMS	Battery Management System
CA*	Cooperative A*
CTC	Cost To Come
CTG	Cost To Go
EDT	Euclidean Distance Transform
ESL	Electronic Systems Laboratory
FSTO	First Search Then Optimize
GPIO	General Purpose Input/Output
HCA*	Hierarchical Cooperative A*
HPA*	Hierarchical Path-finding A*
KVO	Kinematic Velocity Obstacle
LQR	Linear Quadratic Regulator
MAPF	Multi-Agent Path Finding
MARS	Multi-Agent Robot System
MPC	Model Predictive Control
MRMP	Multi-Robot Motion Planning
ORCA	Optimal Reciprocal Collision Avoidance
PRM	Probabilistic Roadmap
PWM	Pulse Width Modulation
ROS	Robot Operating System
RPP	Randomized Potential Planner
RRA*	Reverse Resumable A*
RTA*	Real-Time A*
RVO	Reciprocal Velocity Obstacle
SDA	Sense, Detect and Avoid

SLAM	Simultaneous Localisation and Mapping
STAIR	STanford Artificial Intelligence Robot
UAV	Unmanned Aerial Vehicle
URDF	Unified Robot Description Format
VO	Velocity Obstacle
WHCA*	Windowed Hierarchical Cooperative A*

Chapter 1

Introduction

1.1. Background

The use of robotic systems has been applied to many domains, including agriculture, manufacturing and medical care (Behmanesh *et al.* 2017, McKinsey & Company 2019, Ni *et al.* 2015). A recent application of robotics that has received much attention is the development of autonomous or self-driving cars. One of the reasons for this was a competition hosted by DARPA called the Grand Challenge, where the competitors were tasked with building an autonomous car that could complete a selected racetrack. This contest occurred in 2004, 2005 and 2007, the latter of which tested the vehicle's ability to navigate in an urban environment (Whitaker 2006).

Two of the key enabling technologies for the eventual integration of autonomous vehicles into commercial transport are autonomous navigation and automatic collision prediction and avoidance. Traditionally, research on automatic collision avoidance for unmanned vehicles has focussed on using on-board sensors such as vision-based sensors, lidar, and radar to predict and avoid collisions. In this thesis, we develop a cooperative navigation system for multiple Autonomous Ground Vehicles (AGVs) where all vehicles communicate their state and intent information, and coordinate their planning to produce collision-free trajectories for all vehicles.

Multi-robot systems have advantages over single-robot systems in that they are capable of performing distributed tasks more efficiently, where a parallel workforce increases task execution speed (Lerman *et al.* 2006). The challenge with developing and implementing a multi-robot system is that it introduces the need for communication and coordination strategies, as resources need to be shared between the robots (Yan *et al.* 2013). The most fundamental of these resources is the space that the robots occupy, as no two robots are allowed to be at the same space at the same time. This is often expressed as the Multi-Robot Motion Planning (MRMP) problem, and has been a research focus since 1985 (Kant and Zucker 1985).

One of the most well known examples of a multi-robot system is the KIVA warehouse management system (Mountz *et al.* 2006), which was later acquired by Amazon Robotics. This system coordinates hundreds of AGVs which are used to transport goods inside a

warehouse. In order for this system to work, there needs to be a multi-robot planning strategy used by the vehicles so that they do not collide with one another.

The solution to the MRMP problem can be approached in either a centralised or decentralised way. The centralised approach is easier to solve, as it does not rely as heavily on communication and coordination between the agents. The drawback to the centralised approach is that it does not scale well to a large number of agents, and introduces the problem of a single point of failure. For this reason, it is often beneficial to use the decentralised approach (Dewangan *et al.* 2017).

This project focuses on the development of a system that is able to coordinate the cooperative navigation of several AGVs in a decentralised way. This includes the development and integration of the necessary algorithms, as well as testing the performance of the system both in simulation and using a practical setup.

1.2. Problem statement

There are several scenarios where it is advantageous for multiple mobile robots to operate in the same environment. One such scenario is an automated warehouse, where AGVs are used to move items around the warehouse. In this scenario, it is important that the vehicles are able to navigate within the warehouse without causing collisions. This is a challenging problem, as the movement of the vehicles creates a highly dynamic environment.

1.3. Goals

A solution to this problem is to create a cooperative navigation system, which allows the vehicles to communicate and coordinate amongst themselves. Using this system, each vehicle can find and execute a collision-free trajectory from its starting position to its goal position, while taking the trajectories of the other vehicles into account. The coordination between the vehicles should be done in a decentralised way, allowing for a more resilient, scalable and modular system. The design of this cooperative navigation systems forms the first goal of this project.

The second goal is that a simulation environment should be created which will allow for the rapid design and testing of each individual subsystem and algorithm, as well as for the complete system. Once the system is proven to work in simulation, the appropriate practical tests should be done to ensure the proper working of the system under real world conditions.

1.4. Autonomous navigation

Autonomous navigation is a complicated and multi-faceted problem, which can be simplified by using an appropriate framework. This allows for decoupling the problem into more manageable sub-problems. Several of these frameworks have been suggested over time and, even though there are some nuanced differences, they mostly follow the same approach.

This approach is often referred to to as the "sense - plan - act" framework (Faigl 2017), and partitions the problem into those three areas. The "sense" component refers to the agent's ability to build a model of the environment using data that it aggregates from sensors. It also uses these sensors to determine the state of the robot, such as where it is in that environment. The second component is responsible for forming a "plan" to accomplish a high level task specification, such as navigating to a designated location. In order to form this plan, it needs to use the model of the environment that it has made as well as knowledge of its own state. The next and final step is to "act" on this plan, which would involve using control systems to actuate motors, moving it in the desired direction.

An example of this framework can be seen in Figure 1.1. This example demonstrates a popular approach to solving the perception problem, which is through the use of Simultaneous Localisation And Mapping (SLAM). One of the challenges of finding both the map of the environment, as well as the pose of the robot in that environment, is that you need the one to find the other. Only once you have a map of the environment can you say where you are in that map. Likewise, when building the map, it is necessary to use information about the pose of the robot to determine the spatial relationships between objects in the environment. This is referred to as the "chicken and egg" problem, and

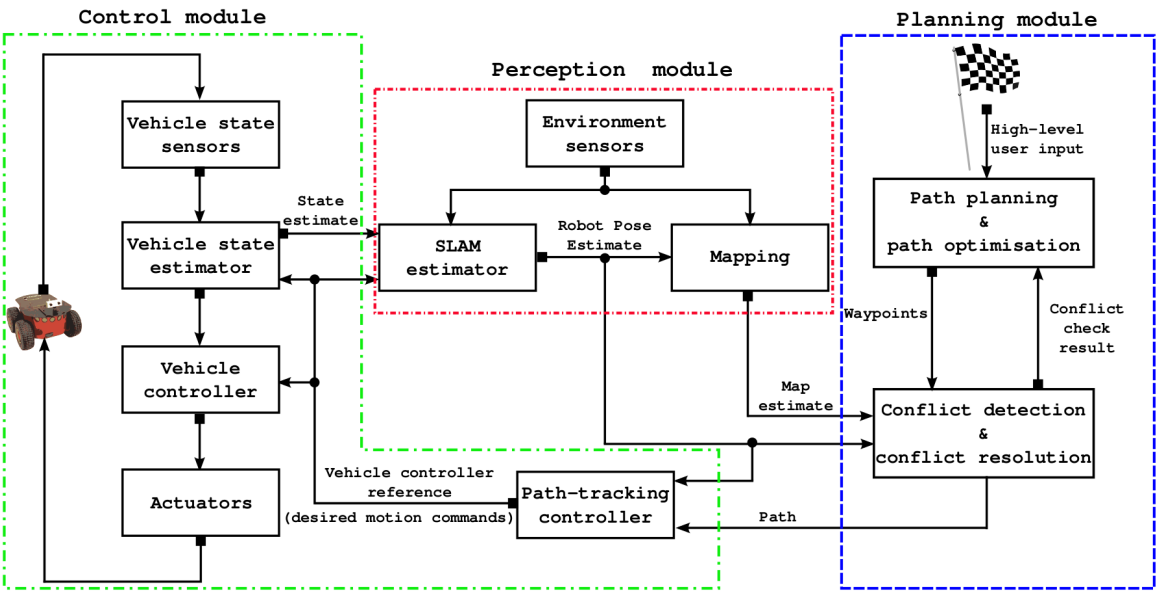


Figure 1.1: The autonomous navigation framework presented by Maseko *et al.* (2020).

led to the development of the SLAM algorithm. SLAM is an active field of research and beyond the scope of this project, but a thorough introduction is presented by Cadena *et al.* (2016). For the purposes of this project it is assumed that a map of the static environment is known beforehand.

Once the map of the environment has been constructed, and the pose of the robot has been determined, it is possible to move to the planning and control phases. The framework in Figure 1.1 divides this into four main components, namely path planning and optimisation, conflict detection and resolution, path tracking and finally the vehicle controller. The first of these two components is responsible for finding a collision free path that the robot can follow to get to its destination. Once this path has been found, it becomes the responsibility of the path tracking controller and vehicle controller to ensure that this path is successfully executed.

An alternative approach can be seen in Figure 1.2, which shows the *move_base* framework used by the Robot Operating System (ROS) middleware platform. Although these two frameworks are mostly the same, there are slight architectural differences. The *move_base* framework divides the planning into a global and local planner. These two planners are hierarchically structured, meaning that the output of the global planner feeds into the input of the local planner. The global planner is responsible for finding a collision free path to the destination, only taking the static obstacles into account. Once this has finished, it hands this path to the local planner, which further optimises the path and ensures that it is executed. In this way, the local planner groups the optimisation of the path with the tracking of the path. When the path contains temporal information then these two components are often referred to as the trajectory planner and the trajectory tracker.

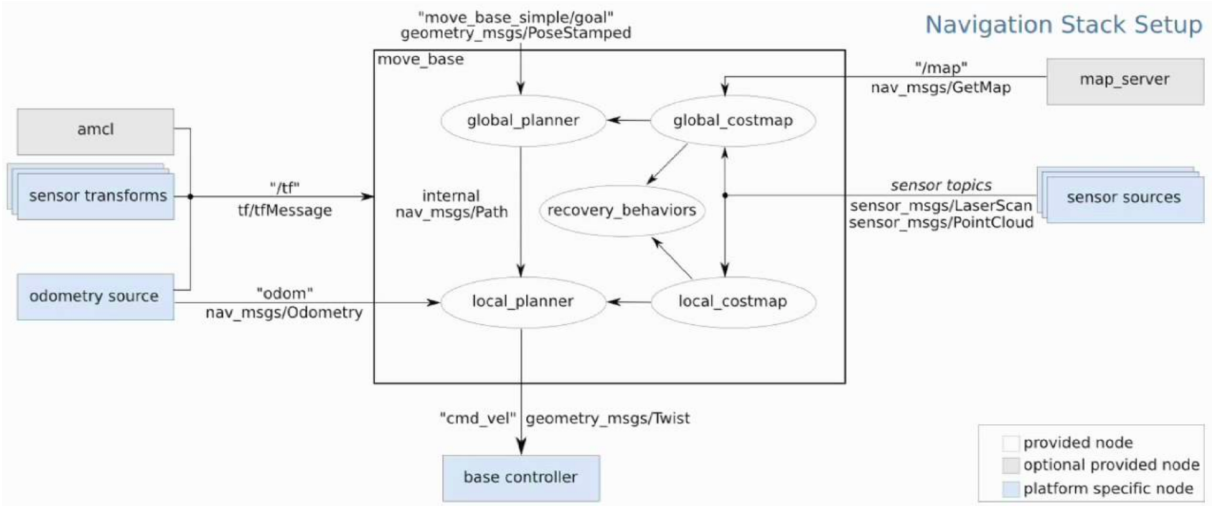


Figure 1.2: The ROS *move_base* package architecture (ROS 2020*a*).

1.5. Objectives

The research goal is broken down into the following research objectives:

1. Develop a cooperative trajectory planning algorithm that can accommodate the trajectories of other vehicles, as well as the kinematic constraints of the vehicle. It should also be able to plan around the presence of static obstacles in the environment.

2. Develop a trajectory tracking module which is able to execute the planned trajectories by controlling the vehicle. It should be able to correct any deviations of the actual trajectory from the planned trajectory.

3. Develop a simulation setup to test the systems and algorithms as they are being developed.

4. Construct a physical test setup with multiple ground vehicles.

5. Implement and test the decentralised cooperative navigation system using both the simulation setup and the practical test setup.

1.6. Overview

This project successfully designed and implemented a system which is capable of co-operatively navigating several AGVs in an environment with static obstacles, using a decentralised planning and communication technique. An example scenario where this can be applied is illustrated in Figure 1.3, showing three vehicles navigating in an environment with static obstacles. Each of the vehicles is responsible for finding and executing a collision-free trajectory from its starting pose to its goal location, while taking the trajectories of the other vehicles into account. The vehicles communicate their planned trajectories with each other, enabling them to coordinate and plan their trajectories in a decentralised way.

The proposed cooperative navigation system was implemented on three different classes of vehicles, as seen in Figure 1.4. When performing the practical tests, it was necessary to design and implement a vehicle pose estimation system. This was done using ArUco fiducial markers and a computer vision detection algorithm. The results obtained when performing the simulated and practical tests can be seen in the YouTube video found at (Viljoen 2020).

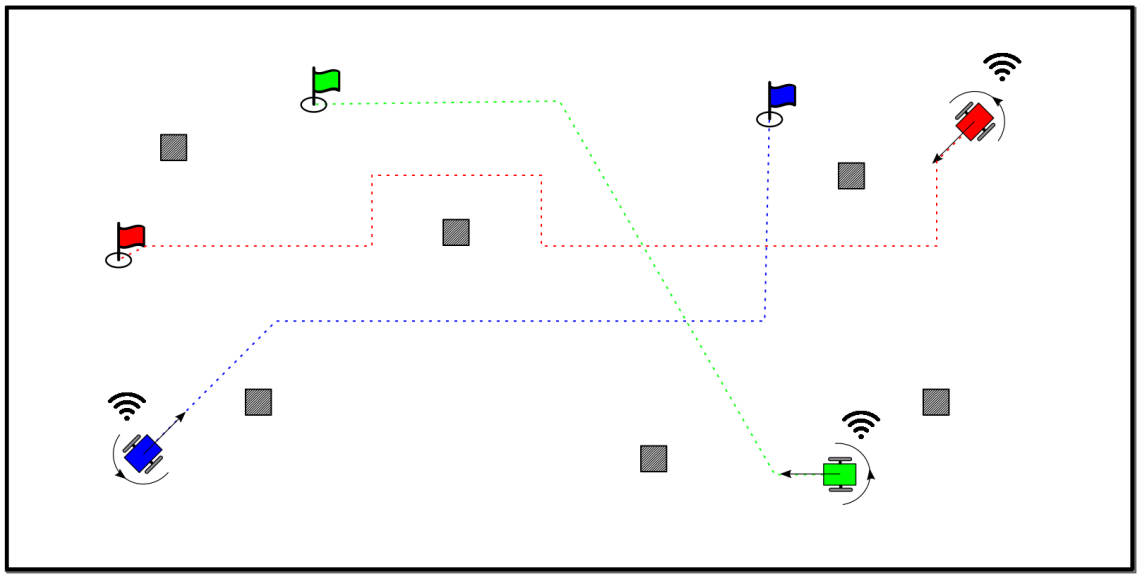


Figure 1.3: An illustration of where a cooperative navigation system can be used to navigate the vehicles from their starting positions to their goal locations. The vehicles are able to prevent collisions by communicating their intended trajectories with each other.

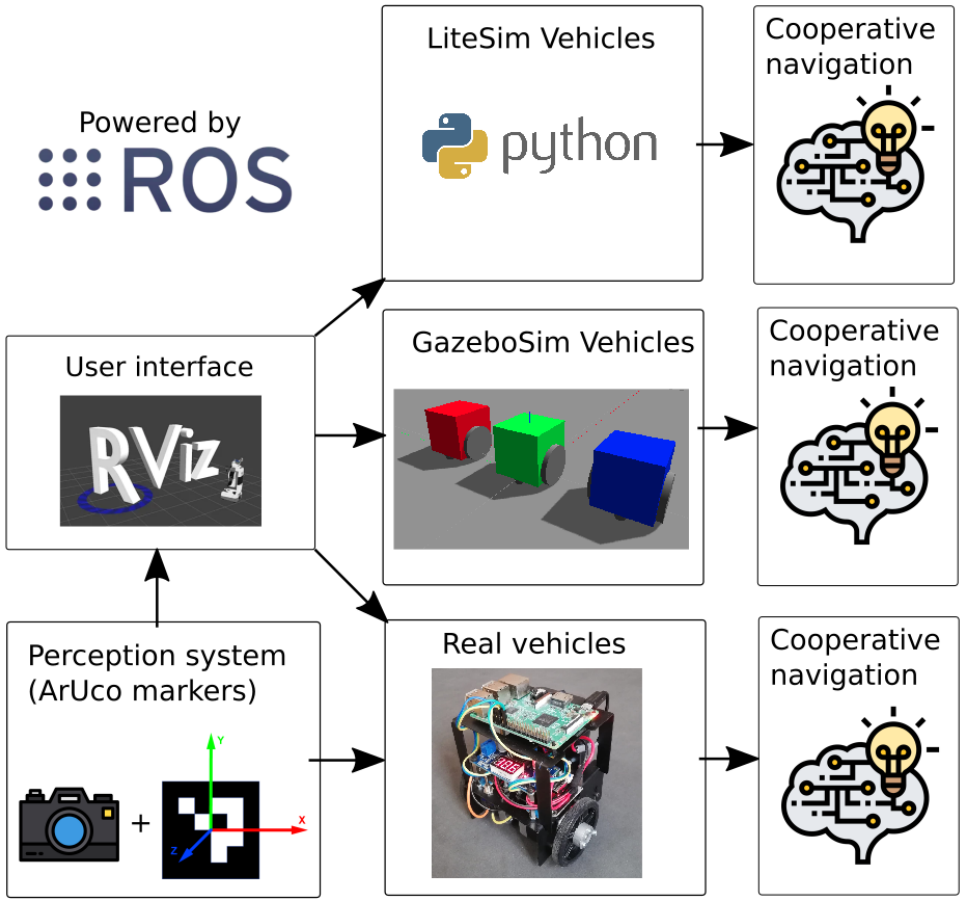


Figure 1.4: Several different kinds of vehicles were used to test the performance of the algorithms under different conditions. These vehicles vary in the extent to which they accurately model the real-world behaviour of vehicles.

1.7. Contributions

The research presented in this thesis makes the following primary contributions:

1. A cooperative navigation and collision avoidance system for multiple AGVs with kinematic constraints was developed and successfully demonstrated both in simulation and using practical experiments.

2. A practical test setup was created consisting of three ground vehicles and an external vision-based pose estimation system. This experimental setup can be used for future multi-vehicle research projects in the Electronic Systems Laboratory (ESL).

1.8. Scope and limitations

The scope of the project was limited in the following ways:

- This project focused primarily on the cooperative trajectory planning and tracking algorithms, and not on the mapping of the environment. This means that the algorithms assume that a map of the static environment is available at all times.

- The vehicles used during this project are ground vehicles with differential-drive mechanical systems, although the algorithms could be extended so as to apply to other vehicles, such as aerial and underwater vehicles.

- The algorithms in this project were tested on a small number of vehicles, with no more than six vehicles used at a time during the simulation tests and three in the practical tests.

- The vehicle's environment consists of only static obstacles and other cooperative vehicles, and not uncooperative dynamic obstacles.

1.9. Thesis structure

The rest of the thesis is structured in the following way:

- **Chapter 2 - Related Work.** This chapter investigates previous approaches that have been used for path planning and path tracking, as well as existing tools that facilitate the development of autonomous navigation algorithms.

- **Chapter 3 - System Overview and Modelling.** This chapter has two distinct parts. The first is the discussion and description of the overall software architecture that was used in this project. It explains how all the various components fit together

to form the complete solution. It also describes the experimental setup that was used for developing and testing the various components of the project. This includes both the simulation setup as well as the hardware implementation. The second part presents the modelling of the vehicle's behaviour as well as the environment in which it operated.

- **Chapter 4 - Cooperative Trajectory Planner.** This chapter presents the detailed design of the cooperative trajectory planner. It explains how the vehicles communicated and coordinated amongst each other, as well as how the trajectories were determined.
- **Chapter 5 - Trajectory Tracking.** This chapter presents the design of the trajectory tracking module. This module is responsible for executing the trajectories that have been planned by the cooperative trajectory planning module.
- **Chapter 6 - Physical Vehicles.** This chapter describes the construction of the physical vehicles that were used for the experimental tests.
- **Chapter 7 - Vehicle Pose Estimation System.** In order to implement the trajectory planning and tracking algorithms, it is necessary for the vehicles to know their pose within the environment. This chapter describes the approach used to find and track the pose of the physical vehicles during the practical tests.
- **Chapter 8 - System Integration and Results.** This chapter describes the integration of the full system and the tests that were performed to evaluate the performance of the system and its various subsystems. The results from both the simulation tests and the practical tests are presented.
- **Chapter 9 - Conclusion and Recommendations.** This chapter presents a summary of the work done, discusses the main findings, and gives recommendations for future research.

Chapter 2

Related Work

The purpose of this chapter is to give a brief historical overview of the field of robotics, as well as provide insight into the different approaches used in literature when designing path finding and path tracking algorithms. It also includes a section describing one of the popular frameworks used to implement these algorithms, called the Robot Operating System (ROS).

2.1. History of robotics

Even though the use of the word "robot" dates back to 1920 (Capek 2020), the first robot only made it's appearance in 1961. This robot, called Unimate (RobotHallOfFame 2020), was developed by General Motors and used on the assembly lines. Its purpose was to take die castings from machines and perform welding operations on automobile frames, which was dangerous and unpleasant work. The success of Unimate led to a wealth of research being done in the field of industrial automation and robotics (Gasparetto and Scalera 2019).

One of the key components of a robotic system is that is should be able to act autonomously, which requires a level of intelligence. This became even more important as the robots were tasked with increasingly complex work. This need for robot intelligence led to interdisciplinary work between the fields of engineering and computer science, where the field of Artificial Intelligence (AI) was only starting to be explored. The use of the term "artificial intelligence" had started just prior to Unimate's appearance, having originated at a conference in Dartmouth in 1956 (Solomonoff 1985, Moor 2006, Kline 2011). Even though there was significant progress in the fields of robotics and AI, most of the work focused on developing specialised skill sets as required by the individual research projects. This fragmentation continued until 1966, when much of the research that had been done was unified around the development of a mobile robot platform called Shakey (Kuipers *et al.* 2017). The goal of this project was to build a general purpose mobile robot that was able to execute high level instructions. It was able to do this by incorporating the research that had been done in the fields of robotics, computer vision and natural language processing.

The development of Shakey represented a shift in the development of robotics, as it necessitated more collaboration between various research groups, and emphasised the interdisciplinary nature that is inherent to the field of robotics. One of the notable outcomes of the project was the development of the A* path finding algorithm by Hart *et al.* (1968), which has formed one of the building blocks of robot AI. This collaborative work has continued, and has resulted in powerful open-source projects such as ROS and Gazebo (Quigley 2009, Koenig and Howard 2004).

One of the major robotics research areas is called collaborative robotics, and focuses on the interaction between robots and other autonomous agents, whether it be humans or other robots. Collaborative robots, or cobots, were first introduced by Colgate *et al.* (1996), and were mainly used in the manufacturing industry. When several robots have to work together to accomplish a goal this is known as a Multi-Agent Robot System (MARS).

Parker and Head (2010) categorises MARS based on their motion coordination being either relative to other robots, relative to the environment, relative to external agents, or a combination of the three. Formation movement, such as vehicle platooning, is an example of motion coordination relative to other robots (Kavathekar and Chen 2011). An example of when the motions are coordinated relative to the environment is multi-agent mapping systems, where the robots work together to quickly and accurately map an environment (Konolige *et al.* 2002). Multi-robot target tracking systems, such as the ones used for surveillance and videography, are examples of MARS where the motion is coordinated relative to an external agent (Jung and Sukhatme 2007).

2.2. Path finding

One of the core components of an autonomous navigation framework is the path finder. This was also the component that related the most to the goal of this project, and therefore a thorough literature study was done so as to better understand how it works.

2.2.1. Path finding paradigms

The goal of path finding is to find a sequence of actions that will result in an agent moving from a starting position to a desired end position. The path finder should output this sequence of actions, as well as the path that will result from executing them. Figure 2.1 illustrates a scenario where this technique can be applied. In the figure there is an agent, in this case a red vehicle, that needs to reach a desired destination, represented by the red flag. The path finder needs to find a path that will allow the agent to reach the desired location without colliding with any of the obstacles.

At this point it is necessary to clarify what is meant by the configuration space of the agent, and how that differs from the workspace of the agent. Figure 2.2 shows an example

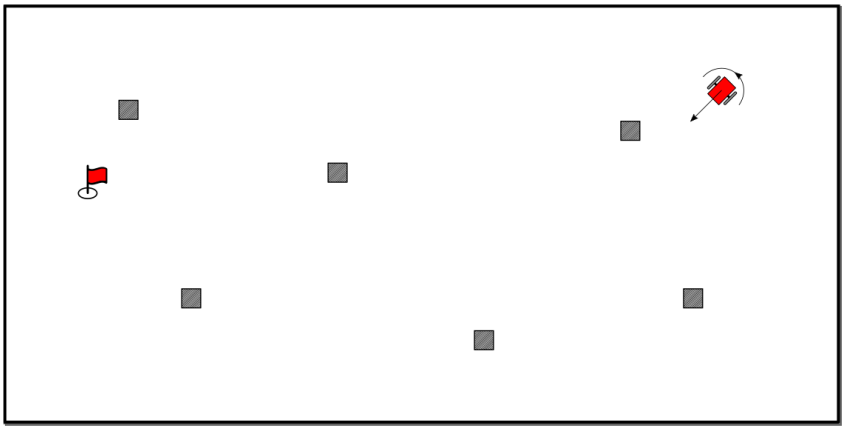
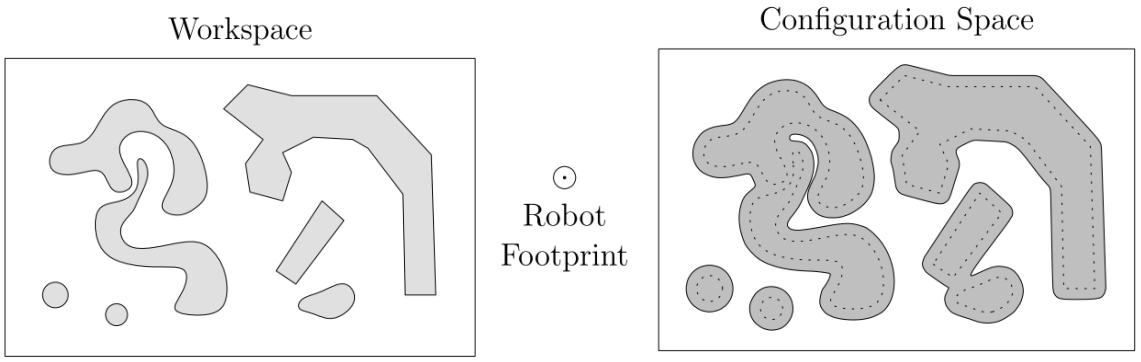


Figure 2.1: An example scenario where path finding can be applied. The red vehicle needs to find a valid path to the red flag.



(a) The workspace of an robot **(b)** The configuration space of a robot

Figure 2.2: The difference between the workspace of a robot and its free configuration space. The obstacles in the workspace have been inflated to accommodate the footprint of the robot (Wooden 2006).

workspace for a robot, which is the unobstructed regions within the robot's environment. The configuration space is similar to this, except that is has been inflated to take the footprint of the robot into account. It is necessary to make this distinction so as to prevent the agent from choosing a path that comes too close to an obstacle, which would cause it to enter into a collision.

Path finding has been the subject of much research since 1968, when the A* algorithm was first developed by Hart *et al.* (1968). Since then many improvements have been suggested, as well as alternate ways of approaching the path finding problem. What follows is a discussion on some of the most popular approaches for path finding, as well as how the algorithms can be enhanced to make them more effective and efficient.

Brute force search

Brute force search has two main components, node generation and node testing. The algorithm generates new nodes using the action space of the agent, and then tests the

validity of the new node by checking to see if it is in the collision-free configuration space of the agent. If the node is valid, it is added to a list of nodes that needs to be further explored. It does this repeatedly until it reaches the desired end node. One of the most common implementations of brute force search is an algorithm called breadth first search, an example of which is shown in Figure 2.3. The node exploration is shown at three different points, with Figure 2.3 showing all the nodes that were explored by the time the search completed.

One of the problems that arise almost immediately is a phenomena known as *combinatorial explosion*. This is where the amount of nodes to be explored grows exponentially as the path finding progresses, as shown in Figure 2.3. The uniform node exploration in all directions is what causes the combinatorial explosion. The effect of this phenomena worsens as the dimensionality of the search space increases, crippling the usefulness of the algorithm.

Informed search algorithms address this problem by adding a suitable heuristic to guide the order in which nodes are explored. A common implementation of this is to assign a cost to each node, which is then used to prioritise the sequence in which nodes are expanded. This cost of a node can be calculated as follows:

$$f(n) = g(n) + h(n) \tag{2.1}$$

where $f(n)$ is the total cost associated with the node, $g(n)$ is the cost-to-come for that node, and $h(n)$ is the heuristic. The cost-to-come of a node is equal to the cost-to-come of its parent node plus the cost of the action it had to execute to reach that node. The heuristic can be chosen in one of several different ways, although the most common would be to use the Euclidean or Manhattan distance from that node to the end destination. For this reason the heuristic is often referred to as the cost-to-go for that node. This implementation of the path finder will always return the optimal path, provided the chosen heuristic is optimistic, meaning that it always underestimates the cost-to-go.

The most well known version of the informed search is A*, an example of which is shown in Figure 2.4. From this figure it can be seen that the number of explored nodes is drastically less than that of breadth first search, making the algorithm significantly

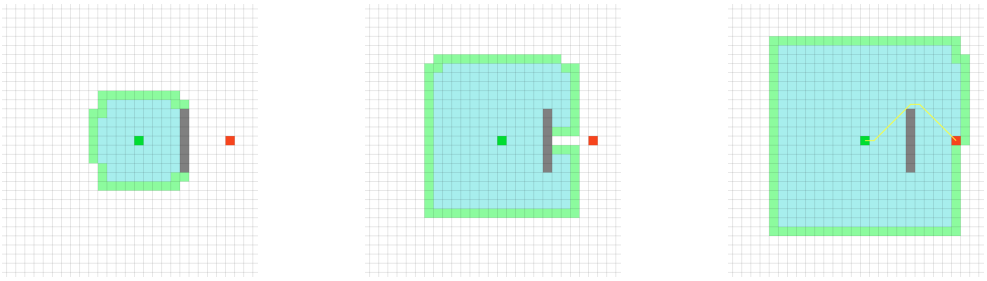


Figure 2.3: Breadth first search node expansion at three different points, the last one being when the search has terminated. (Xu 2020).

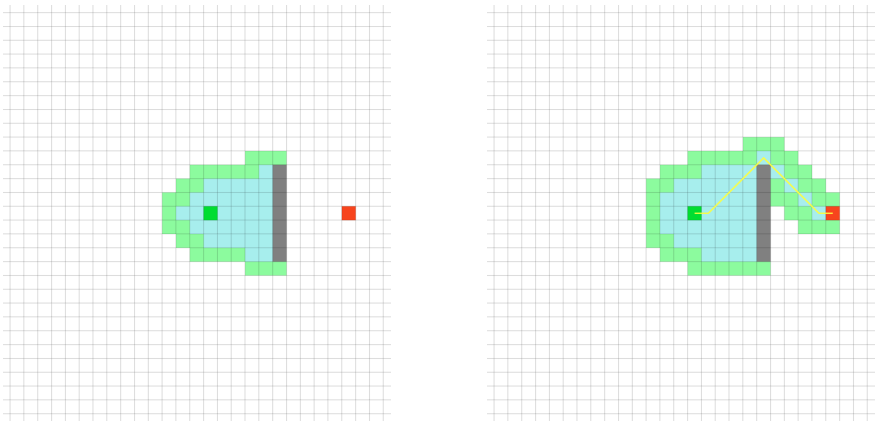


Figure 2.4: A* node expansion at two different points, the last one being when the search has terminated. (Xu 2020).

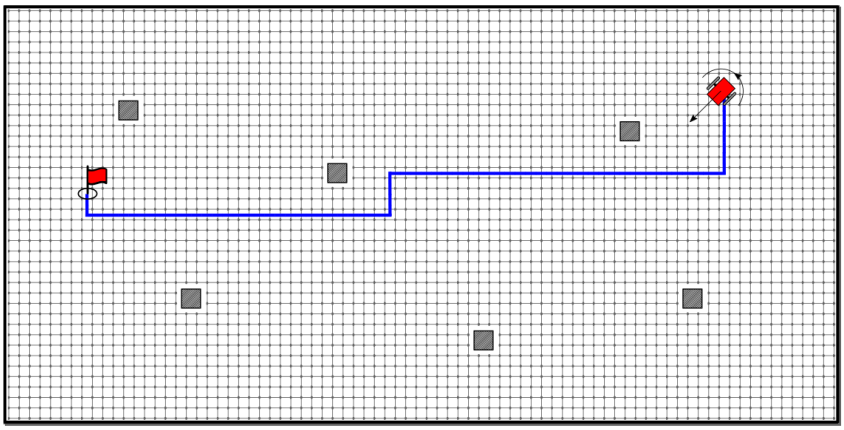
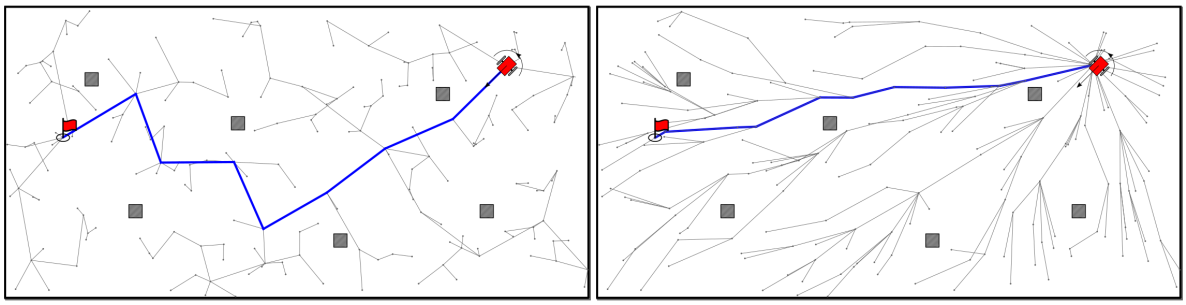


Figure 2.5: An example solution that a path finding algorithm such as breadth first search or A* would yield.

more efficient. By prioritising the exploration of nodes closer to the goal node, rather than uniformly exploring nodes, the total number of explored nodes is less than for breadth first search. Figure 2.5 shows an example path that the path finder will yield if it is correctly applied.

Sampling-based search

An alternative path finding approach is the use of sampling-based techniques. Sampling-based planning first proposed to address motion planning of a six degrees of freedom robot arm (Donald *et al.* 1987). Sampling-based planning outperforms grid-based approaches when working with a high-dimensional search space, as it suffers less from the "curse of dimensionality". The increasing popularity of sampling-based approaches led to the development of the Randomized Potential Planner (RPP) (Barraquand and Latombe 1991). This approach uses a combination of potential fields and random walks, which are used to escape the local minima that usually cripple the usefulness of potential field methods. Kavraki *et al.* (1996) proposed the Probabilistic Roadmap (PRM) algorithm, a



(a) (b)

Figure 2.6: Sampling-based approaches to solving the single-agent path finding problem. In (a) the RRT algorithm is used, where as the extension of this algorithm called RRT* is used in (b).

sampling-based approach that allows for multi-query path finding.

The Rapidly-exploring Random Tree (RRT) algorithm presents a further advance in the field of sampling-based path planning, and has become one of the dominant techniques in the field (LaValle 1998). It is a single-query method that finds a path to the goal region by sampling points and building a tree. An extension of this algorithm was presented by Karaman and Frazzoli (2011) and is called RRT*, which is capable of finding more optimal paths to the goal region. Figure 2.6 shows an example scenario where RRT and RRT* are applied to the single-agent path finding scenario. The path returned by the RRT* algorithm is significantly shorter, making it more optimal.

2.2.2. Path finding extensions

Regardless of the search paradigm used, there are several enhancements that can be applied to a path finding algorithm to make it significantly more effective. Some of these extensions are discussed in the following sections. What follows is not meant to be an exhaustive list, but merely some of the enhancements that are of particular use to this project.

Roadmap

Search algorithms follow a two-step procedure. The first step is to build a graph that accurately represents the agent's environment, and the second step is to perform a search that finds a suitable path through the graph. Building a graph that represents the agent's environment is a computationally demanding process, and contributes significantly to the total time taken for an agent to find a valid path to its destination. This can cause problems, as often times the path finding exercise needs to be repeated frequently, or has to be completed under time constraints. In order to ameliorate the aforementioned problem, the graph that is constructed can be reused for later searches. This is known as a multi-query search algorithm, and works well in an environment that does not change

significantly.

There are several different ways of modelling an agent's environment with a graph, often called a roadmap. Given that the roadmap will be reused several times, it is worth implementing an effective graph representation. Some of the most popular roadmap representations are discussed below.

Cell decomposition Cell decomposition seeks to divide the free configuration space of the agent into non-overlapping regions, called cells. Cell decomposition can be performed in either an exact manner, or as an approximation. Figure 2.7 illustrates how an approximation can be used to construct a roadmap of the environment. In Figure 2.7 (a) and (b) the roadmap is approximated in a uniform fashion using fixed decomposition. The disadvantage with fixed decomposition is that it always under-represents the free space, resulting in potentially viable paths being discarded. This can be partially mitigated by using a finer resolution, but this in turn has the undesirable side effect of increasing the memory usage of the roadmap. Figure 2.7 (c) shows a more advanced technique called adaptive decomposition, which can be used to more accurately represent the free configuration space of the agent, whilst still being memory efficient. Exact cell decomposition can be seen in Figure 2.7 (d), and has the benefit of maintaining the fidelity of the original map, however it can produce unpractical roadmaps.

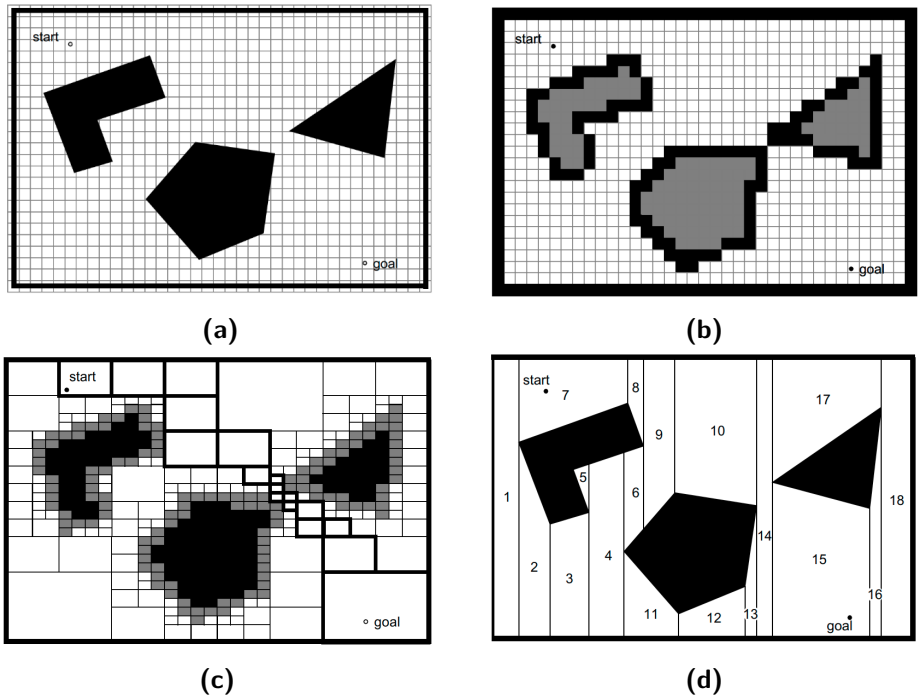


Figure 2.7: In (a) the configuration space is divided into cells, and in (b) each cell is labelled as either free or obstructed. A more advanced approximation technique, called adaptive decomposition, can be seen in (c). Exact cell decomposition is illustrated in (d) (Siegwart and Nourbakhsh 2004).

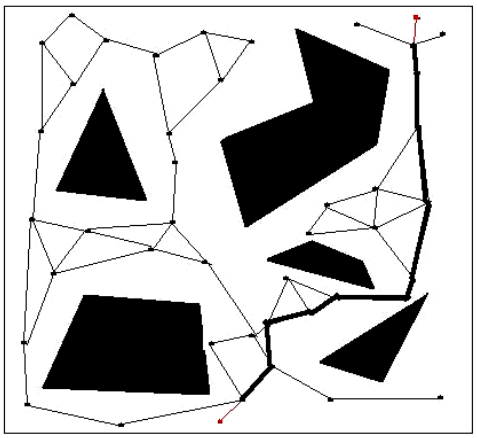


Figure 2.8: An illustration of a probabilistic roadmap (Masehian and Sedighizadeh 2010).

Probabilistic roadmap Another approach to constructing roadmaps is to use a sampling-based approach. The idea behind this is that points in the free configuration space of the agent can be randomly sampled, and then connected to previously sampled points, if valid connections exists. The strength of this approach is that the accuracy of the map can be improved at any stage by sampling more points, allowing for a flexible roadmap representation. The most well known implementation of this approach was published by Kavraki *et al.* (1996) and is called a Probabilistic Roadmap (PRM), an example of which can be seen in Figure 2.8.

Planning in dynamic environments

One of the most pervasive problems in path finding is dealing with uncertainty. This problem often manifests when planning in dynamic environments, where obstacles are non-stationary. One solution to this problem is to replan whenever changes in the environment are detected. This becomes a costly exercise in a highly dynamic environment, and suffers from being highly inefficient. Often there are only minor changes, and most of the original plan can remain intact, with only slight adjustments being made. Stentz (1994) proposed an algorithm called D*, which can reuse previous path finding results, only repairing the path where necessary. This technique is an uninformed search based on Dijkstra's search, and does not make use of a heuristic to guide the search. Stentz (1995) proposed an extension to D*, called focused D*, which incorporates the heuristic used by A*, making it an informed search. Koenig and Likhachev (2005) published a further extension of this algorithm called D* Lite, which focused on making the algorithm execute more efficiently. Another approach to planning in uncertain environments is called Lifelong Planning A*, and is also able to repair previous solutions as the environment changes (Koenig *et al.* 2005).

Taking kinematic constraints into account

One of the challenges with finding suitable paths for robots is that it is often necessary to take the kinematic constraints of the platform into account. One example of these constraints is that the robot might be limited in the velocity or acceleration that it can attain, due to mechanical constraints. Another example is when working with nonholonomic platforms, where the orientation or turning radius of the robot has to be taken into account. This is often the case when planning for AGVs, as they typically have kinematically constrained steering systems.

Accommodating these kinematic constraints has been an area of research since an early stage, pioneered by solutions such as the one presented by Laumond (1987). Laumond *et al.* (2006) presents a overview of some of the most popular approaches used. This work was extend by Hönig *et al.* (2016) so as to work in a multi-agent setting.

Anytime planning

Sometimes it is useful to allocate a length of time that an algorithm has to compute a solution, and then have it return its best result at the end of that allocated time. In this way the optimality of the solution can be balanced with the time it takes to execute. This class of algorithms is known as anytime algorithms, and has been applied to many different domains (Boddy and Dean 1989). An implementation of this for path finding was developed by Likhachev *et al.* (2004), called Anytime Repairable A* (ARA*). This algorithm was extended so as to work in dynamic environments, and was published as Anytime Dynamic A* (ADA*) by Likhachev *et al.* (2005). ADA* is a result of combining two path finding algorithms, ARA* and D* Lite.

ARA* works on the premise that greedy path finding algorithms find paths more quickly, although yielding sub-optimal paths. A* can be made more greedy by inflating the cost-to-go heuristic, as is illustrated in Figure 2.9.

As a result of inflating the cost-to-go heuristic, the path length increases, but the number of operations decreases, indicating a commensurate decrease in computing time. The path length and number of operations for each heuristic weight used in Figure 2.9 can

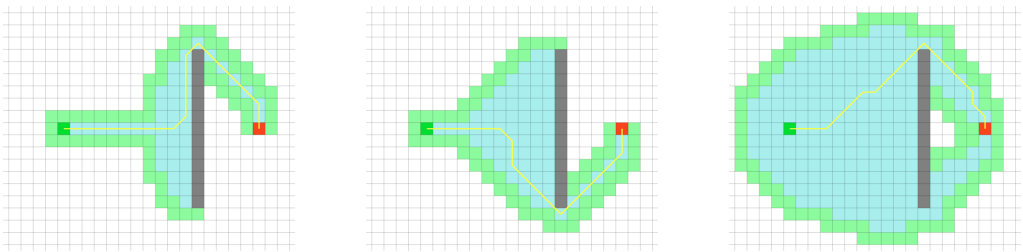


Figure 2.9: Qualitative effect of inflated heuristic on A*. The weighting of the heuristic is decreased in the scenarios going from (a) to (c). (Xu 2020).

be found in Table 2.1.

From Figure 2.9 it can be seen that the first three steps are the same regardless of the heuristic weight used. ARA* makes use of this result by first doing a greedy search, which completes more quickly, and then uses the result to start the execution of the path. While the path is being executed, it repeats the search, but with a less greedy heuristic. This finds a more optimal path, which is then used from that point on. It continues doing this until the optimal path is found, or the overall search is ended.

Informed heuristic

As previously mentioned, one of the problems with using an exhaustive search algorithm is that it can lead to combinatorial explosion. This occurs when the number of nodes that need to be explored grows exponentially as the search progresses. A* partially addresses this problem by adding a heuristic to guide the search, decreasing the amount of unnecessary nodes that are explored. Provided that the heuristic used is optimistic, meaning it is always less than or equal to the actual cost-to-go at that node, the search will yield the optimal path.

The best heuristic to use is the one that can perfectly model the actual cost-to-go from any node, called the true heuristic. Finding the true heuristic will always require at least as much operations as finding the shortest path, and it therefore doesn't make sense to use it unless it can be reused at a later stage. The goal is to find a heuristic that is as close as possible to the true heuristic, whilst not detracting from the overall efficiency of the path finding exercise. In other words, the effort saved by using the heuristic should outweigh the effort used to find the heuristic. One of the ways in which these heuristics can be found is using hierarchical planning, which is discussed in the next section.

Hierarchical planning

The idea behind hierarchical planning is to first perform a search in an abstract search space of lower dimensions than the intended search, and then to use the result of that search to better inform the intended search. Hierarchical A* was first introduced by Holte *et al.* (1998), and presents a technique that can be used to generate suitable heuristics for path finding. This was extended to Hierarchical Path-finding A* (HPA*) by Botea *et al.* (2004), which was specifically aimed at the game design industry. HPA* differs from

Table 2.1: Quantitative effect of inflating heuristic for A* for scenarios shown in Figure 2.9.

Heuristic weight	Path length	Number of operations
10	25.31	169
3	24.14	209
1	22.97	515

hierarchical A* in that it is primarily concerned with finding suitable map representation topologies that can be used to perform searches at various levels of abstraction.

Hierarchical path-finding can be demonstrated with the following illustration. Imagine a car has to find a path between two major cities. This problem can be broken into three parts as follows. The first part is to determine a path to the nearest highway, after which the path is found to the off-ramp closest to the destination city, and then lastly the path is found from the off-ramp to the desired location in the city. In this way the path finding exercise is significantly simplified and as soon as the path to the highway is found the execution of the path can commence, while the rest of the path is planned. The underlying technique at play here is the use of abstraction to simplify planning. Once the map has been abstracted to cities and highways, that layer of abstraction can be used to decrease the planning time. Using this technique does not necessarily yield the optimal path, but can yield a near-optimal path if the level of abstraction is chosen well.

Cooperative planning

Silver (2005) published a paper that presented a technique called Cooperative A* (CA*). This technique suggests the use of a reservation table to facilitate cooperative path finding. The agents involved plan in sequence according to predetermined priorities, and share the paths that they compute with each other. The shared paths are then recorded in the agents' reservations tables, which are used when they conduct their own planning. Even though the use of this method can produce collision-free paths for all the agents involved, there are scenarios where the path finding will fail despite the fact that valid solutions exist. This is due to the fact that each agent plans in a greedy fashion, not taking into account agents that need to plan after it. If there are bottlenecks in the environment then the lower priority agents might not get access through it due to it always being used by a higher priority agent.

CA* incorporates the information from its reservation table to determine where the other agents are going to be in the future, and then performs a search in the resulting space-time to find the best path. Searching in space-time means that the dimensionality of the search is increased, resulting in longer searching times. This is addressed through implementing a hierarchical search strategy called Hierarchical Cooperative A* (HCA*). Firstly the agent conducts a spatial search, ignoring the information from the reservation table. This search is called Reverse Resumable A* (RRA*), and searches backwards from the destination so as to find the accurate spatial cost-to-gos that can be used as a heuristic for the space-time search. In this way, the space-time search only has to navigate around the non-stationary obstacles.

One of the challenges that arise with HCA* is that the agent has to calculate a path to the destination in space-time, which is a high dimensional state space. If the destination is far away, this can take some time, delaying the start of the agents movement. One way

of addressing this is to use a technique that interleaves planning and execution, such as Real-Time A* (RTA*) presented by Korf (1990). It is able to do this by making use of a fixed planning horizon. Although this technique is designed for single-agent searches, the use of the planning horizon can be implemented for HCA* as well. Only planning a fixed window into the future means that the search is limited in its depth, and doesn't have to take collisions into account which might in fact not occur. The resulting algorithm is called Windowed Hierarchical Cooperative A* (WHCA*).

2.3. Multi-agent path finding

The goal of the Multi-Agent Path Finding (MAPF) system is to find collision free paths for all of the agents involved. It must consider potential collisions with both the static and dynamic obstacles in the environment, and usually tries to minimise a cost function. This cost function can be the total distance travelled, total time taken, or some other metric. It can also be a weighted sum of several cost functions. Figure 2.10 shows an example solution that such a MAPF system could output. The most popular approaches to solving the MAPF problem are discussed in the following sections.

2.3.1. Uncooperative planning

Most of the uncooperative planning approaches are based on the Sense, Detect and Avoid (SDA) framework (Chand *et al.* 2018). Using this framework, the predicted trajectory of each of the other vehicles can be represented as a Velocity Obstacle (VO), a concept

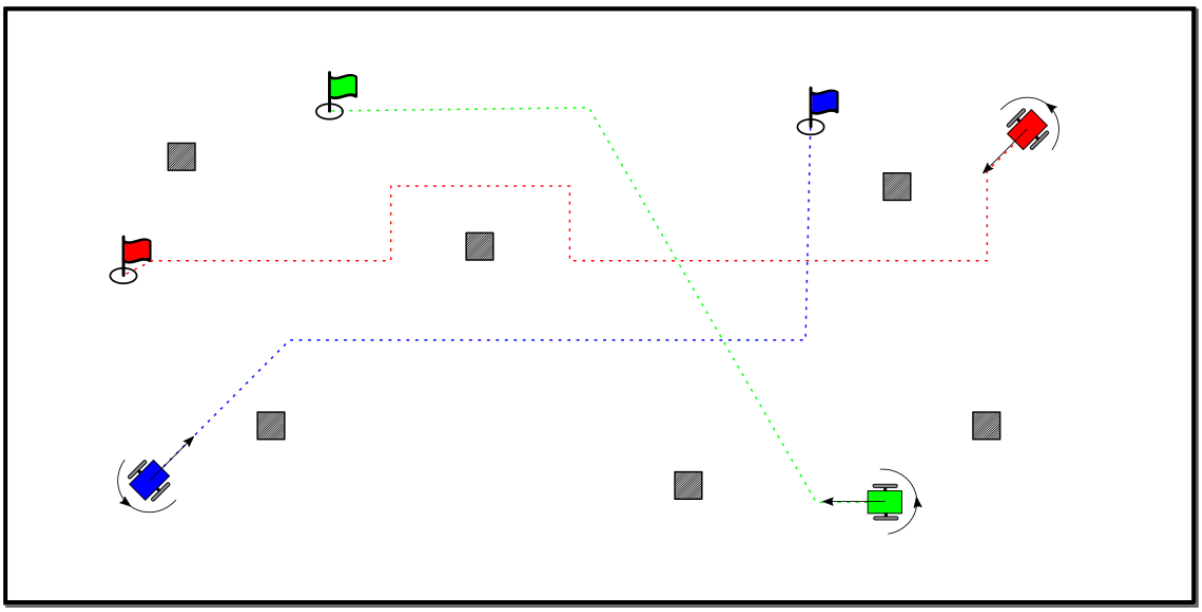


Figure 2.10: An example solution for a MAPF problem, where collision-free paths have been found for all the vehicles in the environment.

presented by Fiorini and Shiller (1998). The VOs of the other agents can then be used in the planning phase to find a collision-free trajectory. An extension of this concept, called the Reciprocal Velocity Obstacle (RVO), relies on the fact that the other vehicles will be using the same approach, allowing for a more accurate prediction of their trajectories (Van Berg *et al.* 2008). Jenie *et al.* (2014) presented an implementation of the SDA approach using VOs for the safe navigation of UAVs.

The RVO has also been extended to use the Acceleration-Velocity Obstacle (AVO), which accommodates varying accelerations (Van Den Berg *et al.* 2011*a*). Wilkerson *et al.* (2014) presented the concept of a Kinematic Velocity Obstacle (KVO), which further improved the idea of the VO by allowing the kinematic constraints of the vehicles to be taken into account.

The benchmark uncooperative planning technique for multi-agent systems is known as Optimal Reciprocal Collision Avoidance (ORCA), and was first presented by Van Den Berg *et al.* (2011*b*). This technique is similar to the RVO, but is able to guarantee smooth trajectories for all the vehicles. ORCA-DD is an extension to this technique that is able to take differential-drive kinematic constraints into account (Snape *et al.* 2010).

One of the disadvantages of using the VO approach is that it relies heavily on the accurate estimation of both the position and velocity of the other vehicles. This is a non-trivial task, but can be mitigated by using a communication protocol, where the vehicles share their positions and velocities with each other. One example of where this communication protocol is implemented can be found in (Godoy *et al.* 2016). More recently, the use of data-driven learning methods to better predict the trajectories of the other vehicles has gained traction. Long *et al.* (2018) presented a reinforcement learning method that works successfully in simulation. This was extended by Fan *et al.* (2020) so as to work for a practical setup, once the *sim-to-real* problem was addressed. The *sim-to-real* problem arises when models that were trained in simulation perform poorly when used in a practical setup as a results of the simulation environment not representing the practical environment accurately.

2.3.2. Rule-based planning

Another simple approach used in MAPF scenarios is rule-based planning. This is how the current traffic system works, where a set of predetermined rules guides the decision making of the agents involved. For example, all of the agents should drive on a particular side of the road, or should yield to traffic coming from a particular side at intersections or roundabouts. There are small variations in these rules depending on the country in which they are implemented, but the rules are consistent within that country. As long as all the agents follow the rules, conflicts should be avoided.

The largest flaw in this approach is that there exists many situations where following

the predetermined rules results in locally sub-optimal solutions. An example of this would be having to wait at a red traffic light, despite there being no other cars in the vicinity. Another disadvantage to rule-based planning is that it can only handle the finite set of scenarios for which it was designed. Rules-based planning may not handle unanticipated scenarios well. A limitation in the ability of the agents to communicate amongst each other is often what necessitates a rule-based MAPF approach, as is the case with the current traffic system. Communication between vehicles on the road today is limited to using indicator lights, brake lights and the occasional hand signal. One of the advantages of implementing driverless cars is that the computers responsible for navigating the cars would be able to harness much more effective means of communication, such as the emerging V2V communication systems (Arena and Pau 2019).

2.3.3. Centralised planning

Another popular approach to MAPF is to implement a centralised planner. This planner is responsible for planning the paths for all the agents involved. The advantage to this approach is that the planner has a complete knowledge of all the desired destinations of the agents, and can therefore find a globally optimal solution. Furthermore, the planner also has guaranteed completeness, which means that it will find a solution if one exists.

The biggest drawback to this approach is that it scales poorly with respect to the number of agents involved in the MAPF exercise. The reason for this is that the planner uses the joint action space and configuration space of the agents, which grows exponentially as the number of agents involved increases. This approach is often thought of as forming a composite agent from all the agents involved, and performing a single search for that composite agent.

2.3.4. Prioritised planning

Prioritised planning aims at addressing the scalability problem that centralised planning faces. It does this by assigning a priority to each agent, and then planning in sequence according to the assigned priority. Consider a situation where three agents need to find a path using prioritised planning. The agent with the highest priority plans first, ignoring the presence of the other two agents. Once the first agent is done planning, it shares its plan with the other two agents. The agent with the second highest priority then plans, taking into account the plan of the first agent, making sure to avoid a collision with it. It then also shares its plan with the other agents, at which point the last remaining agent starts planning its own path. The last agent, which has the lowest priority, attempts to find a plan that is not in conflict with either of the first two agents' plans. If it is successful then all three agents have paths that they can follow which will cause any potential collisions to be avoided. An example of such a MAPF technique is called CA*

(Silver 2005). CA* makes use of a reservation table to store the plans of the vehicles as they complete their planning in the prioritised sequence.

Prioritised planning scales linearly with the number of agents involved in the MAPF exercise, which is a significant improvement on the exponential scaling of the centralised planner. Additionally, due to the decoupled nature of the planning, it is possible to distribute the processing task, allowing for a decentralised implementation. Unfortunately it has to sacrifice both its optimality and completeness guarantees in order to work. What this means is that it can fail to find a global solution, even if one exists. Additionally, the plans that it determines are most often sub-optimal, especially for the vehicles that have low priorities. This is due to the fact that the agents involved are greedy in their search, always finding the best path for themselves, regardless of the agents that have to plan after them. This causes problems in environments where there are bottlenecks.

2.3.5. Conclusion

Choosing the appropriate MAPF technique is dependent on the application, as both the centralised and prioritised approaches have different strengths and weaknesses. As a rule of thumb, the centralised planner works well for scenarios where there are a small number of agents involved, and they are operating in a cluttered environment. On the other hand, the prioritised planner outperforms the centralised planner when the number of agents increases and the agents are planning in a sparsely populated environment.

2.4. Path tracking

Path tracking algorithms are responsible for generating the necessary commands that will allow a vehicle to adhere to a supplied path. These commands are usually velocity commands, but can also be actuator commands, depending on the control system implementation on the vehicle. In this way, it bridges the gap between the path planner and the control system of the vehicle. Many different algorithms have been suggested that can be used as path trackers, some of which are discussed below.

2.4.1. Geometric path tracking

Maseko *et al.* (2020) gives a thorough overview of the different geometric path tracking algorithms that are used. The most basic of these is the "head-to-goal" algorithms, which breaks up the path into several intermediate waypoints. The vehicle then moves in a straight line towards the nearest of those waypoints. Once it reaches this waypoint, it heads to to the next waypoint. It continues doing this until it reaches the final waypoint, which is also the goal location.

A slightly more complicated version of this algorithm is known as the "follow-the-carrot" path tracker (Wit 2000). This algorithm differs in that it continually updates the waypoint to the next one, regardless of whether the vehicle has reached the waypoint or not. This change was made to address the stop-start behaviour that emerges when using the "head-to-goal" tracker.

The pure-pursuit algorithm extends the "follow-the-carrot" algorithm by taking the kinematic constraints of the vehicle into account, as well as its starting orientation (Amidi and Thorpe 1991). As a result of this, it outputs a curvature that the vehicle can follow to adhere to the path. A further extension of this algorithm is know as vector-pursuit, and has the added ability to also take the goal orientation of the vehicle into account.

2.4.2. Model predictive control

The geometric path tracking algorithms in the previous section work well for simple problems, but struggle to handle more complex scenarios. Recently there has been an increasing interest in the use of Model Predictive Controllers (MPCs) as a means of path tracking. MPCs solve the path tracking problem by transforming it into an optimisation exercise, which is repeated periodically. MPC considers a control horizon, finding the optimal control sequence based on an objective function and a set of constraints, which are allowed to be nonlinear if a nonlinear MPC is used. MPC is based on three key principles. The first of these is the use of a model, which is used to predict the behaviour of the agent when control inputs are applied. The second principle is that a cost function is used to find the optimal set of control inputs. The third principle is that a horizon is used, which keeps receding in time for every iteration (Nascimento *et al.* 2018).

One of the disadvantages of using MPC is that it requires significant computational resources, as the optimisation process must be repeated every control loop iteration. For this reason it has been predominately used for slower processes, where the control loop can be executed at a low frequency. The earliest recorded use of MPC was for the control of chemical processes (Qin and Badgwell 2003), which are relatively slow processes. Autonomous navigation is often characterised by controlling an agent in a highly dynamic environment, which requires a fast control system. As the computational capabilities of embedded computer platforms are improving, MPC is gaining increasing traction as a capable path tracker for robotic systems.

MPC is derived from an optimal control technique called a Linear Quadratic Regulator (LQR), which was first proposed by Kalman (1960). The most significant difference between LQR and MPC is that LQR optimises once over the entire time window, whereas MPC optimises repeatedly over a smaller receding time window. This means that it can produce globally sub-optimal solutions, but is more robust to model uncertainties and disturbances.

In order to use a MPC, it is necessary to formulate the problem as a non-linear program, as shown in Equation (2.2).

$$\begin{aligned} \underset{\bar{x}}{\text{minimise}} \quad & f(\bar{x}, p) \\ \text{subject to} \quad & \bar{x}_{lb} \leq \bar{x} \leq \bar{x}_{ub} \\ & g_{lb} \leq g(\bar{x}) \leq g_{ub} \end{aligned} \tag{2.2}$$

There are three important aspects to this formulation. Firstly, it is important to identify the objective function, $f(\bar{x}, p)$, where $\bar{x}$ is the vector of variables containing the state of the agent and control inputs at each timestep, and p is the weightings used in the objective function. These weightings control the priority that is placed on the individual components of the objective function. An example might be that a higher priority is placed on minimising the effort exerted by the vehicle than on the path adherence, which can be accomplished by assigning a larger weighting to it in the objective function. The second aspect is that the constraints must be specified, which is done using the $g(\bar{x})$ function. These constraints can be nonlinear, providing the appropriate solver is used. Both the state and the constraints can be limited between a predefined range, thereby limiting the possible solution space.

The third important aspect is that the nonlinear program must be initialised with a suitable seed. The choice of which seed to use can greatly affect the performance of the optimiser. When using a seed that is close to the optimal solution, fewer optimisation iterations has to be performed before the optimal solution is found. A popular framework for mobile robotics that takes advantage of this is known as First Search Then Optimize (FSTO), and was presented by (Li and Zhang 2019). In this framework, a graph-based search such as A* is done first to find a near optimal solution. This is then used to "warm start" the optimisation, allowing it to yield the optimal solution more quickly. Another advantage of using a "warm start" approach is that it can prevent the optimiser from becoming stuck in a local minima.

2.5. ROS

Much of the implementation of these techniques have unified around a robotics middleware platform called the Robot Operating System (ROS) (Quigley 2009). It is an operating system in the sense that it provides the services expected of an operating system, such as cross-platform hardware abstraction, message passing between services, and a package manager. ROS was developed for the express purpose of collaboration amongst researchers in the field of robotics, allowing scholars to share and distribute their work in a standardised fashion. One of the stumbling blocks in robotics research is that much time is spent on writing code that has already been previously written by someone else. This happens

Figure 2.11: An overview of the main ROS capabilities (ROS 2020*b*).

because the code is not properly maintained or documented, and cannot be easily shared. ROS ameliorates this problem by providing a standardised platform for writing highly modular software components that can be reused by other researchers working on similar projects. This allows researchers to focus on the components that apply to their specific project, while building on the work done previously by other researchers. Being free and open-source, ROS has a strong community of developers, ranging from robotics enthusiasts to the pioneers in the field. This results in a plethora of resources and documentation being freely available.

The active development of ROS started roughly in 2007, although in many ways it was a continuation of research and development previously done at Stanford. Whereas much research had been done in specific areas of artificial intelligence, most of the work was isolated to a specific field, and had not been integrated into a holistic solution. The STanford Artificial Intelligence Robot (STAIR) was meant to address this problem by unifying much of the work that had been done by applying it to a comprehensive project (Quigley *et al.* 2007). One of the outcomes of STAIR was the initial development of ROS, which then evolved into a stand-alone product. ROS has come a long way since its conception, having matured into a comprehensive robotics software ecosystem. Some of the main features of ROS that pertain to the scope of this project can be seen in Figure 2.11, and are discussed in the following sections.

2.5.1. Communication

ROS follows a microservices architecture, where all of the computation is broken up into highly decoupled modules, called nodes. These nodes can communicate with one another in one of three different ways. Firstly, the most pervasive form of communication is using a publish / subscribe topology, which allows for event-driven asynchronous communication between the nodes.

Secondly, nodes can register services, which can handle synchronous communication between nodes. Lastly, nodes can use action servers to facilitate multi-step communication transactions. These action servers implements a finite state machine to track the progress of the communication transaction. All of these communication protocols make use of standardised message types, called ROS messages.

2.5.2. Tools

As the ROS toolchain has evolved, a host of applications have been developed to help facilitate the design and testing of robotics systems. Amongst these is RViz, a 3D visualisation tool that can be used to visualise the content of ROS messages, as well as interact with the individual ROS nodes. Another is RQT, a Qt-based framework that can be used to construct dashboards and various other user interfaces. ROS also has a suite of tools that can be used to monitor and debug all the communication between individual nodes, as well as measure the computational usage of the nodes.

One of the key advantages of using ROS is its ability to integrate with several different robotic simulation environments. Using a simulation environment allows for rapid prototyping and testing of algorithms. It is also a way of preventing costly mistakes, given the expensive price of robotics hardware. Arguably the most widely used robotics simulator is called Gazebo, which has a wide variety of different sensor plugins, allowing the simulation of real-world sensor feedback with a high degree of fidelity.

2.5.3. Capabilities and ecosystem

Arguably the best part of ROS is the vibrant community that has formed around it. This has led to a large degree of collaboration, with a plethora of different packages that have been developed from all around the world. Being able to distribute these packages in a standardised way has meant that less time needs to be spent reinventing the wheel, and more time can be spent developing novel algorithms.

2.6. Conclusion

After reviewing the different approaches, it has been decided to use the WHCA* algorithm for the cooperative trajectory planning and MPC for the trajectory tracking. The WHCA* has been chosen because it allows for decentralised planning through the use of reservation tables, as well as allowing for a real-time system by using a windowing approach. The WHCA* algorithm will have to be modified so that the kinematic constraints of the vehicles can be taken into account. MPC was chosen for the trajectory tracking as it is able to find trajectories which accommodate the kinematic constraints of the vehicle, as well as being able to recover from small deviations, by repeatedly finding a set of commands for the vehicle for a receding window. The algorithms will be developed and tested using the ROS and Gazebo simulation environment, as well as using a practical setup. Most of the research surrounding multi-agent navigation have been applied to the game design industry. When applied to the field of robotics, the algorithms have largely been verified in simulation. One of the objectives of this project is to evaluate the performance of the multi-agent navigation algorithms when tested using a practical setup.

Chapter 3

System Overview and Modelling

3.1. Software architecture

The purpose of this project is to design and implement a system that can be used to perform cooperative trajectory planning and execution for AGVs, which would allow multiple vehicles to navigate in the same environment without colliding. In order to achieve this goal, a top-down-design bottom-up-implementation approach is used. According to this approach, the first step is to design a high-level software architecture which describes all the various subsystems used to achieve the end result, as well as how the subsystems interact with each other. Once this has been done, the subsystems can be developed and tested individually, and integrated together once they all work well. The overall software architecture is shown in Figure 3.1.

The cooperative trajectory planner determines a valid trajectory for the vehicle to reach its goal without colliding with static obstacles or other vehicles. The planned trajectory is then fed to the trajectory tracker which is responsible for executing the trajectory while compensating for external disturbances and model uncertainty. The trajectory tracker calculates velocity commands that are provided as references to the vehicle's velocity controller. The velocity controller controls the translational and rotational velocity of the

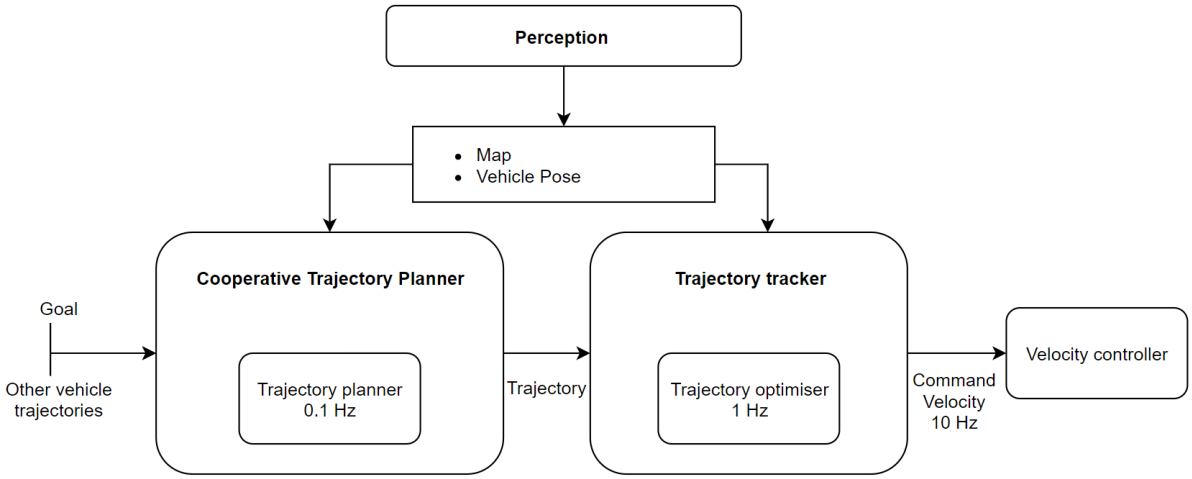


Figure 3.1: The complete software architecture, showing the various sub-components and how they are integrated.

vehicle by actuating the differential drive motors.

One of the goals of the project was to enable the vehicle to perform the cooperative motion planning and execution in a decentralised way. This has numerous advantages, such as being able to leverage distributed computing techniques to improve the performance and scalabilty of the overall system. Additionally, using a decentralised approach mitigates the single point of failure problem, increasing the overall robustness of the system. The architecture shown in Figure 3.1 is implemented on each vehicle. The various subsystems will now be discussed in the following sections.

3.1.1. Cooperative trajectory planner

The goal of the cooperative trajectory planner is twofold. Firstly, it must facilitate the communication and coordination between itself and the trajectory planners running on the other vehicles. In terms of the communication, it must determine what information should be communicated, as well as how often it must be communicated. The cooperative planning technique uses a prioritised planning approach, which means that the vehicles plan sequentially according to their predetermined priorities. This requires a level of decentralised coordination between the vehicles, so that they do not plan at the same time, and in so doing disregard one another's planning. This coordination can be achieved by using a token allocation strategy, where the token is used to determine which vehicle is allowed to plan when.

The second goal of the cooperative trajectory planner is to find a valid trajectory which, when executed, will allow the vehicle to reach its goal without colliding with static obstacles or other vehicles. When planning this trajectory, the kinematic constraints of the vehicle should be taken into account. The trajectory planner finds this trajectory using the following inputs:

- A map which describes the static environment. This is used to plan trajectories which avoid the static obstacles in the environment.
- A table of all the trajectories of the other vehicles. This is used so that it does not plan a trajectory which intersects with any of the other vehicles' trajectories.
- The current location of the vehicle, referred to as the vehicle's pose.
- The desired goal position.

The architecture of the cooperative planner is shown in Figure 3.2. The trajectory planning is executed at regular intervals. After each planning iteration, the vehicle reserves space-time for itself, and communicates the space-time that it reserved to the other vehicles. Even when the vehicle is stationary, it is still moving in space-time, and its stationary

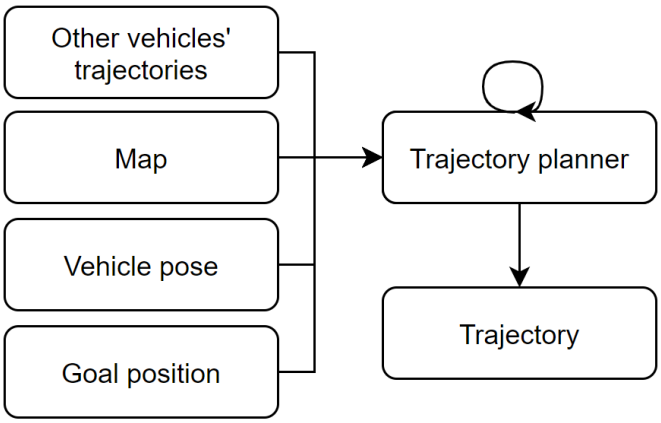


Figure 3.2: Cooperative trajectory planning module architecture.

trajectory must still be communicated to the other vehicles so that they can plan around its reserved space-time.

Two important configuration parameters for the cooperative trajectory planner are the planning frequency and the planning window. It is important that the planning window is sufficiently large so that the vehicle has time to replan before the window ends. A planning frequency of 0.1Hz is used for this project, with a planning window of 30 seconds. This means that the planner replans every 10 seconds for the following 30 seconds. This allows the planner to be more robust, as it still has 20 seconds of reserved space-time left if the planning should fail. It can use this safety margin to reattempt the planning several times, until it finds a valid solution.

The output of the cooperative trajectory planner is a sequence of space-time poses that describes the movement of the vehicle for the next 30 seconds. The trajectory planner does not provide any information about the velocities that the vehicle should execute to adhere to those space-time poses. Calculating these velocities is the responsibility of the trajectory tracking module.

3.1.2. Trajectory tracker

The purpose of the trajectory tracker is to execute the trajectories that are planned by the cooperative trajectory planner. These trajectories specify a sequence of poses that the vehicle should adhere to, with corresponding timestamps. The vehicle therefore plans where it must be and when it must be there, so that it can reach its goal position without any collisions. In order to adhere to these trajectories, the trajectory tracker must meet the following requirements:

- The first and most important requirement is that it should determine a set of velocity commands that can be executed by the vehicle. In other words, it needs to translate the set of space-time poses that it receives into a sequence of velocity commands that when executed will allow the vehicle to reach those space-time poses. The kinematic

constraints of the vehicle should be taken into account when finding the velocity commands.

- Secondly, the trajectory tracker must compensate for small deviations from the desired trajectory. When the vehicle deviates from its desired trajectory, the trajectory tracker must determine the necessary velocity commands to return the vehicle to the trajectory. It should also take the map of the environment into account when doing this, so that it can return to the reference trajectory without colliding with an obstacle.

A well-know approach is to use an optimal control technique called Model Predictive Control (MPC), that uses optimisation techniques to calculate the velocity commands for the vehicle for a time window into the future. MPC formulates the control problem as a constrained optimisation problem with an objective function and constraints. This approach is powerful because it is able to leverage all the advances that have been made in the field of optimisation.

The system diagram for the trajectory tracker is shown in Figure 3.3. The core of the trajectory tracker is an optimisation process that repeats at a fixed interval. This optimisation process takes as input the reference trajectory to which the vehicle must adhere, the pose of the vehicle, and the map of the static environment. The trajectory tracker expects the reference trajectory to be provided in the form of a sequence of future space-time poses to which the vehicle must adhere. The output of the trajectory optimiser is another trajectory. The output trajectory is different from the input trajectory, as it also specifies the velocities necessary to adhere to the trajectory. The velocities calculated by the trajectory optimiser will enable the vehicle to track its desired trajectory, and in so doing reach its goal.

Once the trajectory optimiser has determined the velocities that it needs to execute, it stores them in memory along with their corresponding timestamps. These velocities are then periodically sampled by the velocity sampler, and then sent as references to the vehicle's velocity controller.

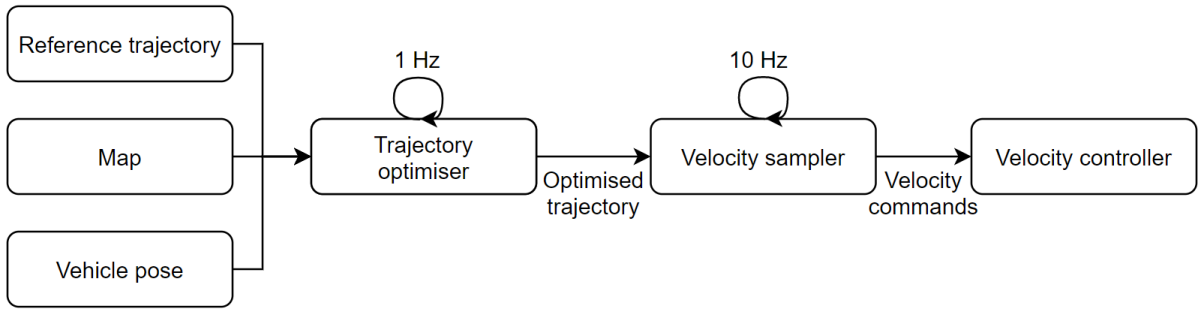


Figure 3.3: Trajectory tracking module architecture.

3.1.3. Perception

The ability of the vehicle to successfully plan and execute trajectories depends on accurate knowledge of itself and its environment. More specifically, the vehicle must know what the static environment around it looks like, as well as where it is in that environment. This is communicated to it by the perception module. Finding a map of the environment, and performing accurate localisation within that map, is a non-trivial task. As such, it has been largely excluded from the scope of this project so that the majority of the time could be spent on the development and testing of the planning algorithms.

This was possible because most of the development of the algorithms was done in simulation, and the simulation program was able to provide both the map of the environment as well as the location of all the vehicles in the environment. It was only when the performance of the algorithms was tested on a practical setup with real vehicles that the map of the environment and the pose of the vehicles were not directly available. For the practical test setup, it was therefore necessary to develop a pose estimation system to determine and track the pose of each of the physical vehicles. The design and implementation of this pose estimation system will be presented in Chapter 7.

3.1.4. Velocity controller

The velocity controller is responsible for listening to the velocity commands that are published by the trajectory tracker. The velocity controller controls the translational and rotational velocity of the vehicle to follow the velocity commands by actuating the differential drive motors.

3.2. Experimental setup

The software and algorithms for this project were developed in an incremental fashion. To facilitate this, a simulation environment was created to test and evaluate the algorithms as they were being developed.

The first iteration of the simulation environment included a simplified model of the vehicle dynamics, written in Python. Once the algorithms performed well on this model, a more sophisticated vehicle model was developed using the Gazebo simulation environment. This Gazebo model of the vehicle included more realistic vehicle behaviour, such as the slipping of the wheels in certain circumstances. This more advanced vehicle model allowed the algorithms to be tested in a more realistic environment. For the final phase of the testing, the algorithms were deployed on physical vehicles, which were built for this purpose. An overview of the experimental setup, showing the different vehicle platforms used to test the cooperative navigation algorithms, is shown in Figure 3.4.

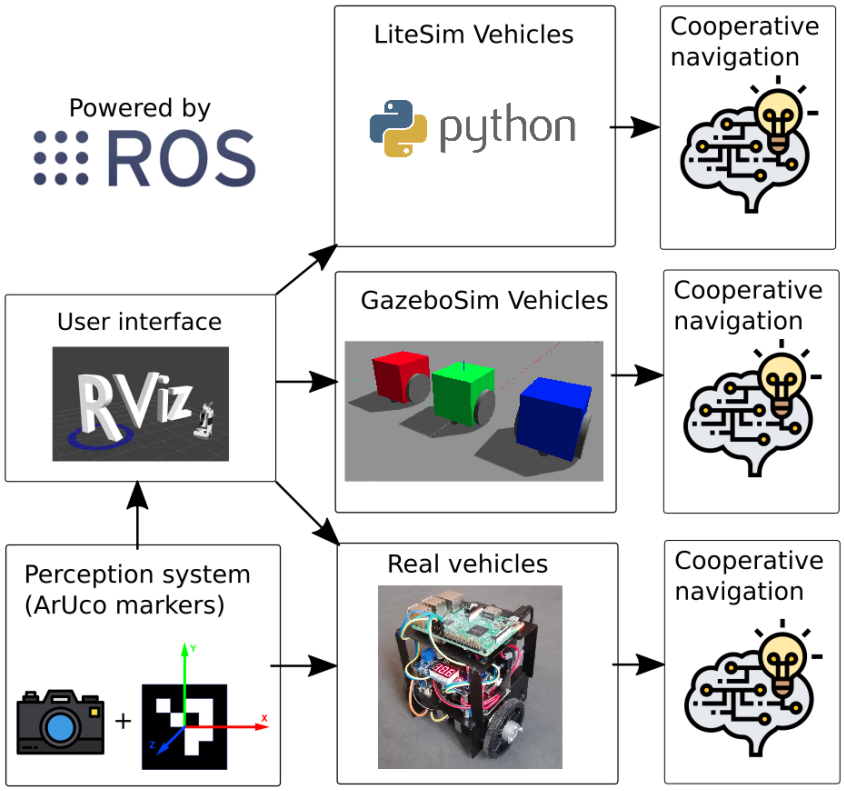


Figure 3.4: Experimental setup, showing the different vehicle platforms used to test the performance of the cooperative navigation algorithms.

The cooperative trajectory planner and the trajectory tracker both require knowledge of the vehicle's pose. Additionally, the vehicle must implement a velocity controller that can execute the velocity commands that are provided by the trajectory tracker. The simplified Python model of the vehicle models the translational and rotational dynamics of the vehicle as first-order systems, and determines the pose of the vehicle by integrating its translational and rotational velocities over time. The Gazebo model of the vehicle includes a velocity controller that controls the vehicle to execute the velocity commands it receives. The Gazebo model also integrates the translational and rotational velocities internally and publishes the pose of the vehicle.

One of the challenges of the project was to construct physical vehicles that could execute the velocity commands that are provided by the trajectory tracker. The design of the physical vehicles that were created for the experimental setup will be presented in Chapter 6. Additionally, a vehicle pose estimation system was developed to determine and track the poses of all the vehicles. The design of the vehicle pose estimation system will be presented in Chapter 7.

3.3. Modelling

Both the cooperative trajectory planner and trajectory tracking modules use a model of the vehicle and its environment when finding a solution. This section describes the process used when modelling the behaviour of the vehicles, as well as the way in which the environment is modelled.

3.3.1. Vehicles

Before modelling the behaviour of the vehicles, it is first necessary to choose which kind of vehicle is to be used for the project. The process used to choose this vehicle type is described below, after which the behaviour of the vehicle is modelled.

Choice of vehicle

One of the design choices that had to be made during the course of the project was which vehicle type to use for the testing of the algorithms. Both the cooperative trajectory planner and trajectory tracker must take the kinematic constraints of the vehicle into account, and these constraints depend on the type of platform that is chosen. The three types of platforms that were considered are shown in Figure 3.5.

The first type of vehicle that was considered is an Ackermann vehicle, which uses an Ackermann steering mechanism as shown in Figure 3.5 (a). This mechanical system was first used by horse-drawn carriages, and is still used in most cars that drive on the roads today. The disadvantage of this mechanical system is that it introduces a turning circle, and is not able to rotate on the spot. Using an Ackermann vehicle therefore requires the use of advanced planning techniques to take the kinematic constraints of the turning circle into account during the planning.

The second type of vehicle that was considered, is an omnidirectional vehicle, as shown in Figure 3.5 (b). Through the use of ingeniously designed wheels, this vehicle is able to translate and rotate in any direction, resulting in almost no kinematic constraints. The

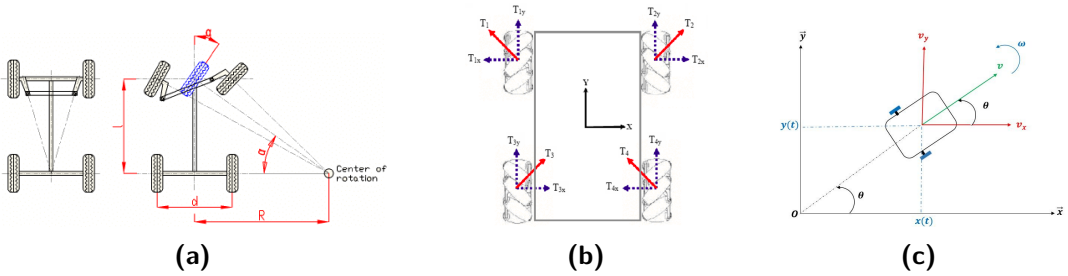


Figure 3.5: Different vehicles considered, with (a) being an Ackermann vehicle (Hrbáček *et al.* 2010), (b) being an omnidirectional drive vehicle (Mohd Salih *et al.* 2006) and (c) being a differential drive vehicle (Mellah *et al.* 2018).

disadvantages of omnidirectional wheels are that they are heavier than normal wheels, are more expensive, and have less traction due to the use of the rollers.

The third type of vehicle that was considered was a differential drive vehicle that uses a differential drive mechanism, as shown in Figure 3.5 (c). This mechanism allows for the wheels to be individually controlled, enabling the vehicle to rotate in one spot by setting the rotational speeds of the two wheels the same but in opposite directions. The ability of the vehicle to rotate in one spot simplifies the planning, as there is no turning radius to take into account. Additionally, this system is also the easiest to build, as there are fewer complicated mechanical parts than either the Ackermann steering vehicle or the omnidirectional vehicle.

After considering all three vehicle types, the differential drive vehicle was chosen as the platform that the algorithms would be tested on. The primary focus of this project was the development and testing of the cooperative planning algorithms, and not the construction of the physical vehicles. The differential drive vehicle type was therefore chosen since its relatively simple design would allow multiple vehicles to be assembled rapidly and with relative ease. However, the trajectory planning and execution algorithms that were developed for this project are not limited to differential drive vehicles only, and could be extended to work with any of the three vehicle types.

Vehicle modelling

Figure 3.6 shows the diagram used when modelling the behaviour of the vehicles. In this figure, the pose of the vehicle is shown at two different time points. The position and

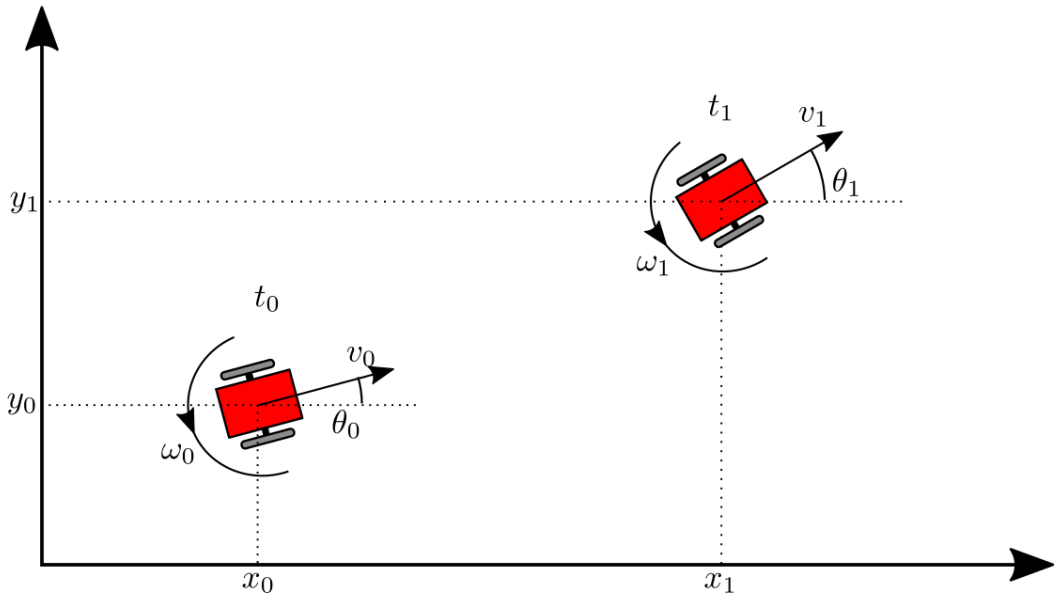


Figure 3.6: Diagram used to perform the mathematical modelling of the vehicles, showing the state of the vehicle at two different time points t_0 and t_1, where $t_1 > t_0$.

orientation of the vehicle is defined by its position (x, y) in the world axis system, and its heading θ relative to the x-axis. The translational and rotational motion of the vehicle is represented by its forward velocity v and its turning rate ω. The vehicle has two wheels: a left wheel and a right wheel, which can be rotated independently, each with its own individual rotational speed. The forward velocity v of the vehicle is determined by the common speed of the two wheels. The turning rate ω of the vehicle is determined by the differential speed of the two wheels. The individual wheel speeds are controlled by the differential drive motors. The continuous-time equations of motion for the vehicle can be expressed as follows:

$$\dot{x}(t) = v_x(t) \tag{3.1}$$

$$\dot{y}(t) = v_y(t) \tag{3.2}$$

$$\dot{\theta}(t) = \omega(t) \tag{3.3}$$

with

$$v_x(t) = v(t)cos(\theta(t)) \tag{3.4}$$

$$v_y(t) = v(t)sin(\theta(t)) \tag{3.5}$$

The forward velocity and the rotational velocity of the vehicle are related to the translational velocities at the contact points between the wheels and the ground

$$v(t) = \frac{v_r(t) + v_l(t)}{2} \tag{3.6}$$

$$\omega(t) = \frac{v_r(t) - v_l(t)}{L} \tag{3.7}$$

where $v_r(t)$ and $v_l(t)$ are the translational velocities of the right and left wheels at the contact points with the ground, and L is the perpendicular distance between the wheels. The translational velocities v_l and v_r of the wheels are related to the rotational velocities of the wheels through the radius of the wheels.

$$v_l(t) = r\omega_l(t) \tag{3.8}$$

$$v_r(t) = r\omega_r(t) \tag{3.9}$$

where w_l and w_r are the rotational velocities of the left and right wheels, and r is the

radius of the wheels.

The next step is to discretise the continuous-time equations of motion to produce difference equations that can be used by the planner and the tracker.

$$x[k+1] = v_x[k]\Delta t + x[k] \tag{3.10}$$

$$y[k+1] = v_y[k]\Delta t + y[k] \tag{3.11}$$

$$\theta[k+1] = \omega[k]\Delta t + \theta[k] \tag{3.12}$$

with

$$v_x[k] = v[k]cos(\theta[k]) \tag{3.13}$$

$$v_y[k] = v[k]sin(\theta[k]) \tag{3.14}$$

and

$$v[k] = \frac{v_r[k] + v_l[k]}{2} \tag{3.15}$$

$$\omega[k] = \frac{v_r[k] - v_l[k]}{L} \tag{3.16}$$

and

$$v_l[k] = r\omega_l[k] \tag{3.17}$$

$$v_r[k] = r\omega_r[k] \tag{3.18}$$

The differential drive motors are commanded at a fixed sampling rate with a sampling period Δt. The commanded wheel rotational velocities are held constant for the duration of a sampling period.

3.3.2. Modelling of environment

Modelling the environment can be decoupled into two different parts. Firstly, the static map of the environment has to be modelled, which can be done using the representation shown in Figure 3.7, where the black tiles represent occupied space and the white tiles represent free space. Using this approach, the static environment is discretised into a grid of cells, with values of either one or zero, where one represents a cell that is occupied and zero represents a cell that is unoccupied. This grid of cells can be stored in memory as a lookup table, which can then be used by the trajectory planning and tracking algorithms.

Typically in an autonomous navigation system there would be a module responsible for finding this map of the environment, but for the purpose of this project it was assumed that this map was provided beforehand, and that the map of the static environment does

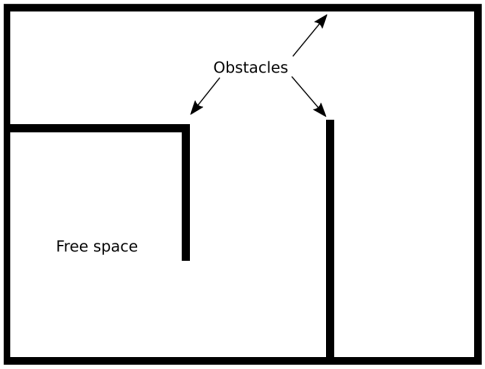


Figure 3.7: Representation used to model the static environment.

not change over time. This way of representing the environment does not allow for taking uncertainty in the static map into account, as all the cells are either permanently occupied or unoccupied. One of the design choices that will have to be made when using this approach is what resolution to use when discretising the environment, as this will influence the performance of the system. A low resolution will limit the complexity of the maps that can be modelled, whereas a high resolution will result in more memory being used when storing the look-up table representation of the map. A higher resolution map will also mean that more time will be used when searching for a valid trajectory, as there are more cells to explore. Choosing this discretisation resolution is done during the design of the cooperative trajectory planning algorithm, and can be found in Chapter 4.

The second component necessary when modelling the environment is to choose a way of representing the dynamic environment. The complete environment of the vehicle can be thought of as the superposition of the static and dynamic components of the environment. For this project, the dynamic environment consisted only of the other cooperative vehicles' trajectories, as uncooperative vehicles and other dynamic obstacles were not included in the scope of this project. When modelling the trajectories of the other vehicles, it is necessary to use a space-time occupancy space representation, as the trajectories of the other vehicles contain both spatial and temporal information.

Figure 3.8 shows four different ways of modelling obstacles using the space-time representation. Both the bounded velocity and bounded acceleration representations are used when there is uncertainty in the trajectories of the other vehicles. This is usually the case when the trajectories of the other vehicles have to be inferred from sensory information. As the cooperative planning approach used in this project expects that the vehicles share their trajectories with each other, these trajectories can be represented using the deterministic path approach.

Once the static and dynamic environments have been modelled, it is possible to form the complete space-time occupancy model. The first step is to represent the static environment using the space-time model, which can be done by simply extending the static occupancy along the time axis, as shown in Figure 3.8. The dynamic environment must then be

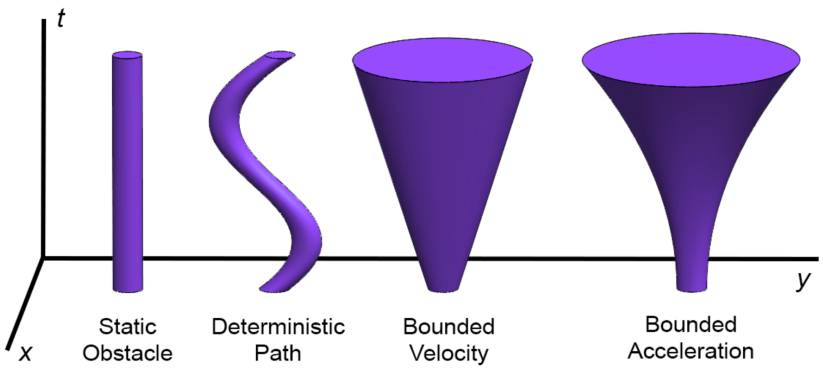
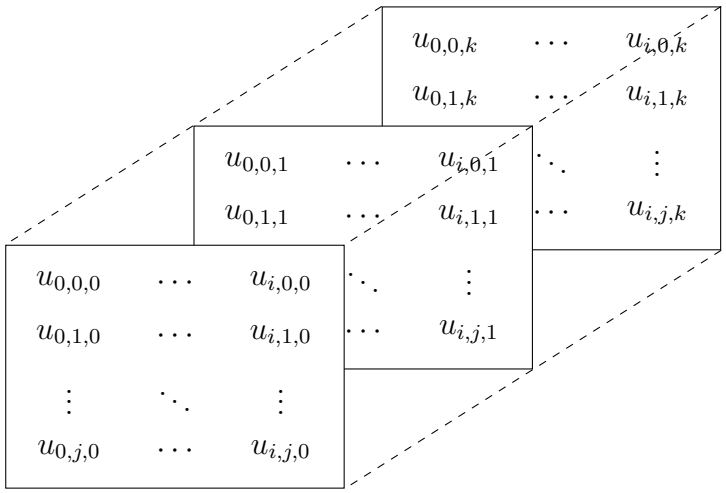


Figure 3.8: Representation used to model the dynamic environment (Pendleton *et al.* 2017).

discretised according to the same resolution as the static environment, so that it can be superimposed onto the space-time representation of the static environment. The result of this process is a three dimensional occupancy representation which can be stored in the following way:



where

$$u_{i,j,k} \in \{0, 1\} \tag{3.19}$$

with $u_{i,j,k}$ representing the occupancy at the position (x, y) and time t, where x, y and t can be found in the following way:

$$x = \frac{i}{x_{\text{res}}} \tag{3.20}$$

$$y = \frac{j}{y_{\text{res}}} \tag{3.21}$$

$$t = \frac{k}{t_{\text{res}}} \tag{3.22}$$

where x_{res}, y_{res} and t_{res} are the spatial and temporal discretisation resolutions.

3.4. Summary

This chapter presented the overall software architecture used for this project, as well as the experimental setup that will be used to evaluate the performance of the system. Furthermore, the modelling of both the static and dynamic environment, as well as the dynamics of the vehicles used, is described in this chapter. The decentralised cooperative navigation system was divided into four major components to be developed in this project, namely the cooperative trajectory planning module, the trajectory tracking module, the physical vehicles, and the vehicle pose estimation system. The design of these components will be presented in Chapters 4, 5, 6 and 7 respectively.

Chapter 4

Cooperative Trajectory Planning

The purpose of this chapter is to present the detailed design of the cooperative trajectory planning module used by the vehicles. The design of this module can be partitioned into two separate parts, namely planning and coordination. The planning component is responsible for finding a valid trajectory from the current pose of the vehicle to its goal position, taking into account the map of the environment as well as the trajectories of the other vehicles.

The coordination aspect of the module is responsible for keeping track of the other vehicles' trajectories, as well as communicating the planned trajectory of the vehicle to the other vehicles. It must also ensure that the vehicles plan in such a way that they take one another's trajectories into account, with no two vehicles planning at the same time. All of the coordination between the vehicles must be done in a decentralised way, so as to allow for a more scalable system.

4.1. Planning

The requirement of the trajectory planning module is that it should be able to find a trajectory to the goal location that prevents collisions with both the static environment as well as the reserved trajectories of the other vehicles. This trajectory should also accommodate the kinematic constraints of the vehicle. This requirement was met by using an enhanced version of the A* algorithm. This section starts by introducing the A* algorithm, and then discusses the enhancements that were added to meet the desired requirement.

4.1.1. A* algorithm

The A* algorithm was first presented by Hart *et al.* (1968), and has been thoroughly documented in several books since then, arguably the most well known one being *Planning Algorithms* by LaValle (2006). The purpose of this section is not to provide exhaustive details on the operation of the algorithm, as this has already been done in *Planning Algorithms*, but rather to highlight the main concepts and features.

Before looking at the algorithm as a whole, it is first necessary to define the concepts and notation used. A graph-based search algorithm uses a *graph* G, which consists of a set of *nodes* $\{n_i\}$ and a set of *edges* $\{e_{ij}\}$, where the edge e_{ij} is from node n_i to node n_j. Each edge has a corresponding cost, $\{c_{ij}\}$. It is often the case that the graph is not available beforehand, but is rather constructed as the algorithm progresses. This is made possible by using the *successor operator* Γ, which receives as input the node n_i and generates the pairs $\{n_j, c_{ij}\}$. The successor operator is often referred to as the *action space*.

The A* algorithm is concerned primarily with finding the *optimal path* from some *start node* $s \in \{n_i\}$ to a *goal node* $t \in \{n_i\}$. It does this by iteratively applying the successor operator to nodes starting with the start node s until the goal node t has been reached, thereby expanding the search to explore new nodes. The node from which the successor operator is applied is referred to as an *explored node*, whereas the nodes generated by the successor operator are known as *visited nodes*. An *open list* is used to keep track of the nodes that have been visted, while a *closed list* is used to record the nodes that have been explored and which are not the goal node. One of the challenges that immediately arises is that the successor operator has to choose which node to explore next. This choice has a measurable impact on the tractability of the algorithm, as the time taken for the algorithm to complete is commensurate to the number of nodes that are explored during the search. An evaluation function $f(n)$ is used to choose which nodes to explored next, and is composed of two parts:

$$f(n) = g(n) + h(n) \tag{4.1}$$

where $g(n)$ is the actual cost of the optimal path from the start node s to node n and $h(n)$ is the cost from the node n to the goal node t. The cost $g(n)$ is often expressed as the Cost To Come (CTC), while the cost $h(n)$ is often expressed as the Cost To Go (CTG). Note that $f(s) = h(s)$ is the cost of the optimal path, and that $f(n) = f(s)$ for all the nodes on the optimal path. Furthermore, $f(n) > f(s)$ for any node not on the optimal path. Unfortunately the values of $g(n)$ and $h(n)$ are not known *a priori*, but can be estimated as $\hat{g}(n)$ and $\hat{h}(n)$ to give $\hat{f}(n)$. A good choice for $\hat{g}(n)$ is to use the smallest cost found so far for the path from node s to node n (Hart *et al.* 1968). This is an estimate because it is possible that at a later point in the search a shorter path from s to n can be found. Finding the estimate $\hat{h}(n)$ of $h(n)$ is more complicated, and there are several different candidate functions that can be considered. Some of these candidate functions are explored in the later sections of this chapter, for now it is sufficient to know that a candidate for $\hat{h}(n)$ is considered *admissible* if it underestimates the actual value of $h(n)$. An admissible function is one which guarantees that the optimal path will be found if it is used.

Now that all the necessary concepts and notations have been introduced, an overview

of the algorithm can be given. The algorithm consists of the following steps:

1. Add s to the list of open nodes which need to be explored, and find $\hat{f}(s)$.
2. Select the open node n which has the smallest $\hat{f}$ value, unless t is one of the open nodes, in which case t should be selected.
3. If the selected node is t, terminate the algorithm as the goal node has been reached.
4. Remove n from the list of open nodes, and apply the successor operator. Find $\hat{f}$ for each successor of n, and mark as open all the successors that are not already marked as closed. If any of the successors of n that have already been closed have a lower $\hat{f}$ than previously calculated, remark that node as open. Go to step 2.

The only part which remains is to extract the path once the algorithm completes. As the nodes are sequentially explored, each node is assigned a reference to its parent node. In this way, the path can be extracted by starting at the end node and simple traversing backwards along the linked list of nodes, until the starting node is reached. An example implementation of the A* algorithm is shown below:

Listing 4.1: A* algorithm pseudocode.

```
closedNodes = []
openNodes = [startNode]
g(startNode) = 0
h(startNode) = heuristicFunction(startNode)
f(startNode) = g(startNode) + h(startNode)
while openNodes not empty:
  n = openNode with lowest f()
  if n = goalNode:
    extractPath from n
    return
  else:
    // Apply successor operator
    for action in actionSpace:
      m = n + action
      g(m) = g(n) + actionCost
      h(m) = heuristicFunction(m)
      f(m) = g(m) + h(m)
      if m not obstructed:
        if m in closedNodes or openNodes
        and g(m) less than previously calculated:
            add m to openNodes again with new f(m)
        else:
          add m to openNodes
  add n to closedNodes
```

4.1.2. Enhancing the A* algorithm

In the following sections, the enhancements that were added to the A* algorithm are discussed. These enhancements turned the base A* algorithm into a cooperative trajectory planner.

Moving in cardinal directions

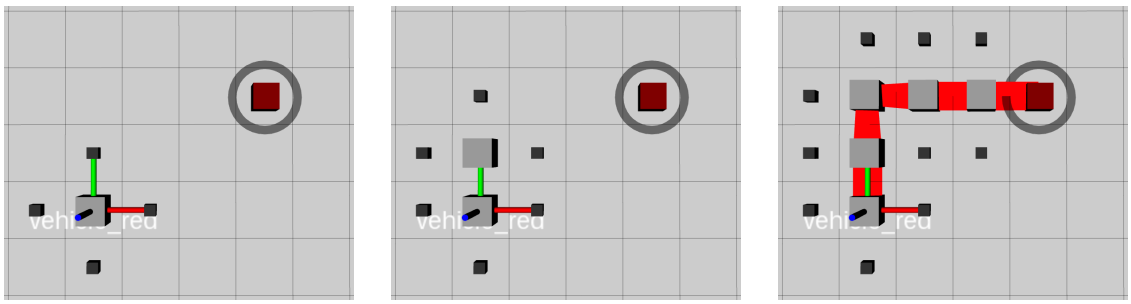
For the first version of the A* algorithm, only the cardinal directions were considered. This meant that the action space of the vehicle was limited to moving forward, backward, left and right:

$$[\Delta x, \Delta y] \in \{[1, 0], [0, 1], [-1, 0], [0, -1]\} \tag{4.2}$$

Figure 4.1 shows a scenario where the cardinal A* algorithm is applied. The nodes that are visited are marked with a small black square, whereas the nodes that are explored are marked with a larger dark grey square. The (x, y) state of the start node s was chosen as (10,4), and that of the goal node t was chosen as (13,6). In (a), the first four nodes are explored, and their states and costs are recorded in Table 4.1.

The CTG of the nodes were determined using the Manhattan heuristic function, which is calculated as follows:

$$\hat{h}(n) = |x_n - x_{end}| + |y_n - y_{end}| \tag{4.3}$$



(a) **(b)** **(c)**

Figure 4.1: Example showing the A* algorithm with only cardinal directions used.

Table 4.1: Initial expansion of A* in cardinal directions.

No.	x	y	CTC	CTG	Total Cost
0 (start node)	10	4	0	-	-
1	11	4	1	4	5
2	10	5	1	4	5
3	9	4	1	6	7
4	10	3	1	6	7

The CTC was calculated using the sum of the actions applied to reach the node, as follows:

$$\hat{g}(n) = \Sigma u_k \tag{4.4}$$

where u_k is the k^{th} action applied, and can be found using the following equation:

$$u_k = \sqrt{\Delta x_k^2 + \Delta y_k^2} \tag{4.5}$$

The total cost was calculated using an equally weighted sum of the CTC and CTG, as follows:

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n) \tag{4.6}$$

where $\hat{f}(n)$ represents the estimated total cost, and $\hat{g}(n)$ and $\hat{h}(n)$ represents the estimated CTC and CTG respectively. As an example, the cost of node 1 can be calculated using the above equations:

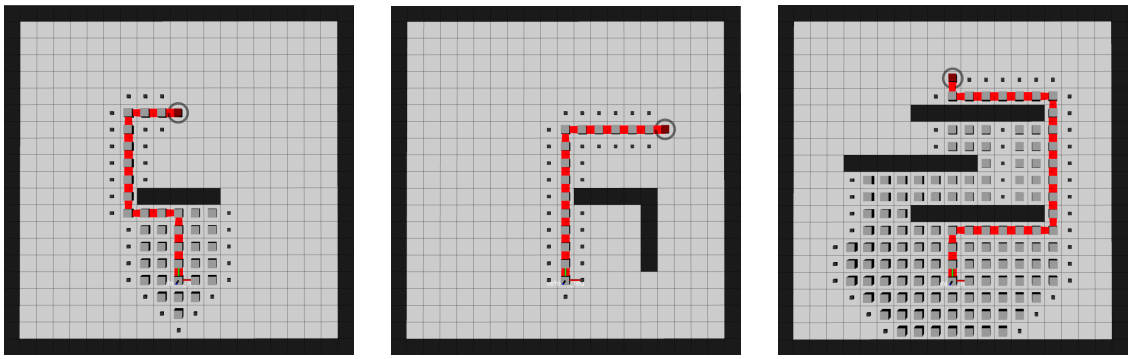
$$\hat{h}(n) = |11 - 13| + |4 - 6| = 4 \tag{4.7}$$

$$\hat{g}(n) = \sqrt{1^2 + 0^2} = 1 \tag{4.8}$$

$$\hat{f}(n) = 1 + 4 = 5 \tag{4.9}$$

Node 1 and 2 are next highest in priority, seeing as they have the lowest total cost. They have the same total cost however, so a tie breaker had to be used to decide which one was to be explored next. The chosen tie breaker was that the node which was visited most recently has the higher priority. This resulted in node 2 being explored next, as can be seen in Figure 4.1 (b). Figure 4.1 (c) shows all the nodes that were visited and explored during the path finding exercise, as well as the final path that was returned by the algorithm.

In Figure 4.2, the algorithm's ability to find a path around obstacles is demonstrated. The obstacles are marked with black tiles. When visiting the neighbouring nodes, the ones that were obstructed failed the admissibility test, and were therefore not added to the list



(a) **(b)** **(c)**

Figure 4.2: Path finding around obstacles using A* in cardinal directions.

of open nodes. As can be seen from Figure 4.2, the presence of obstacles greatly increases the amount of nodes that are explored, depending on the location of the obstacles. In Figure 4.2 (b) the obstacle is between the goal and destination, but because of the way in which the nodes are explored, the presence of the obstacle does not affect the planning, or the final path that is returned.

Adding diagonal directions

Limiting the actions of the agent to movements only in the cardinal directions results in paths which are longer than necessary. Shorter paths can be found by allowing the agent to also move diagonally, in the intercardinal directions. This can be achieved by expanding the action space of the agent as follows:

$$[\Delta x, \Delta y] \in \{[1,0],[1,1],[0,1],[-1,1],[-1,0],[-1,-1],[0,-1],[1,-1]\} \tag{4.10}$$

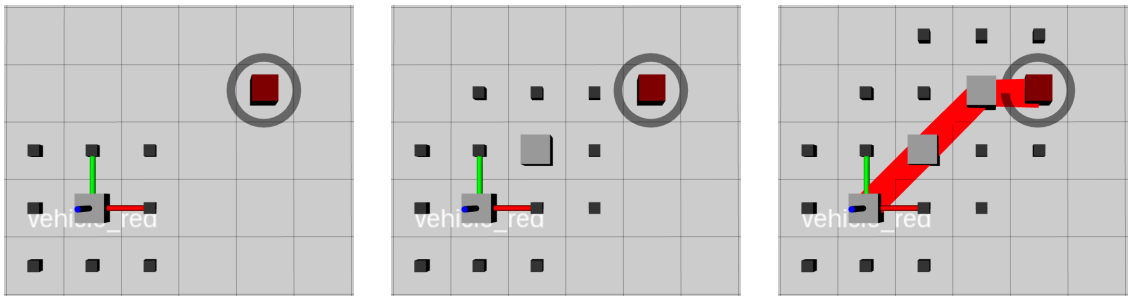
There are now 8 actions that may be applied. A similar path finding exercise as before can now be repeated using the extended action space. The results are shown in Figure 4.3. As was expected, the resulting path is considerably shorter.

Table 4.2 shows the first eight nodes that were explored, with their corresponding states and costs. The CTG was determined using the Euclidean heuristic function, which can be calculated using the following equation:

$$\hat{h}(n) = \sqrt{(x_n - x_{end})^2 + (y_n - y_{end})^2} \tag{4.11}$$

The CTC was again calculated using the sum of the actions applied to reach the node, as follows:

$$\hat{g}(n) = \Sigma u_k \tag{4.12}$$



(a) **(b)** **(c)**

Figure 4.3: A* in cardinal and diagonal directions.

Table 4.2: Initial expansion of A* in cardinal and diagonal directions.

No.	x	y	CTC	CTG	Total Cost
0 (start node)	10	4	0	-	-
1	11	4	1	2.83	3.83
2	11	5	1.41	2.24	3.65
3	10	5	1	3.16	4.16
4	9	5	1.41	4.12	5.54
5	9	4	1	4.47	5.47
6	9	3	1.41	5	6.41
7	10	3	1	4.24	5.24
8	11	3	1.41	3.61	5.02

where u_k is the k^{th} action applied, and can be found using the following equation:

$$u_k = \sqrt{\Delta x_k^2 + \Delta y_k^2} \tag{4.13}$$

The total cost was calculated using an equally weighted sum of the CTC and CTG, as follows:

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n) \tag{4.14}$$

where $\hat{f}(n)$ represents the estimated total cost, and $\hat{g}(n)$ and $\hat{h}(n)$ represents the estimated CTC and CTG respectively. As an example, the cost of node 2 can be calculated using the above equations:

$$\hat{h}(n) = \sqrt{(11-13)^2 + (5-6)^2} = 2.24 \tag{4.15}$$

$$\hat{g}(n) = \sqrt{1^2 + 1^2} = 1.41 \tag{4.16}$$

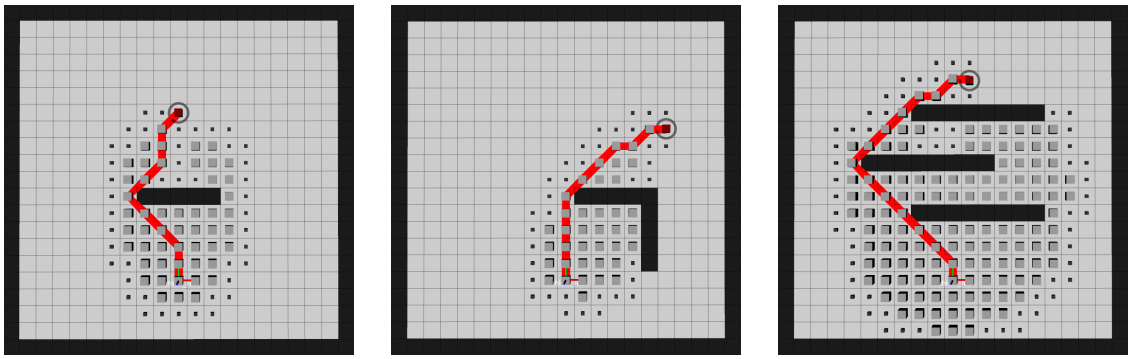
$$\hat{f}(n) = 1.41 + 2.24 = 3.65 \tag{4.17}$$

According to Table 4.2 the next node that should be explored was node 2, which had the lowest cost. This is indeed the case, as can be seen in Figure 4.3 (b).

The obstacle avoidance exercises were also repeated using the new extended action space. The results are shown in Figure 4.4. In all three scenarios the paths that were returned were indeed shorter than when the Manhattan distance was used as the cost function and the action space was limited to only the cardinal directions. An interesting observation is that in all three cases, the total number of explored nodes was more. This is especially true in the case of Figure 4.4 (c), where the obstacle now lies in the way of the shortest path, and a path around it must be found.

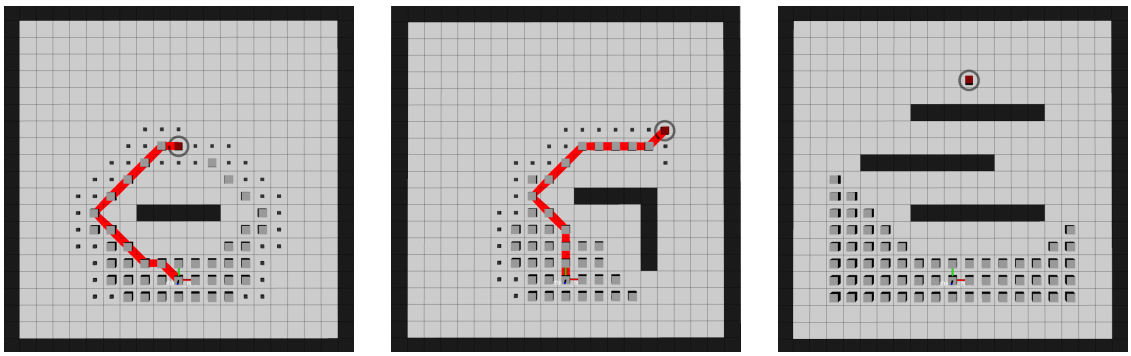
Adding vehicle footprint

One of the necessary components of the path finding module is that it should be able to take the footprint of the vehicle into account when finding the path. In order to do this, an inflation radius is introduced. The purpose of the inflation radius is to accommodate



(a) (b) (c)

Figure 4.4: Path finding around obstacles using A* in cardinal and diagonal directions.



(a) (b) (c)

Figure 4.5: Path finding around obstacles using A* in cardinal and diagonal directions while taking vehicle footprint into account.

the fact the vehicle occupies more than one tile, which means that the vehicle must be more than one tile away from the obstacles at all times. This is accomplished by adapting the algorithm so that it checks whether the new node does not intersect with any of the obstacles or with any of their neighbouring nodes. The path finding exercises was then repeated with this new adaptation and a footprint of two tiles, the results of which are shown in Figure 4.5.

One of the consequences of adding the footprint is that it further restricts the number of admissible nodes, and by implication also reduces the number of admissible paths. This can result in no viable paths being found, as is the case in Figure 4.5 (c).

Planning in space-time

One of the requirements for the planner is that it must to be cooperative. The cooperation was implemented by having each vehicle publish a message stating its planned trajectory (position versus time) for a fixed time window into the future. The planner that has been developed up to this stage only finds and returns a path in space, but without any temporal information. In order to address this shortcoming, an additional dimension was added to the search space to represent the temporal information. Using this new search

space representation, spatiotemporal planning could be performed.

Adding temporal information to the search space results in a trajectory being returned rather than a path, the distinction being that a trajectory contains temporal information whereas a path does not. This means that the path planning module has been upgraded to a trajectory planning module, which is a step closer towards the required specification.

One of the challenges encountered when planning in a spatiotemporal state space is visualising the temporal information. Fortunately the spatial components of the planning are limited to two dimensions, as a result of working with ground vehicles. This means that the third spatial dimension is not yet used, and can be used to visualise the temporal information. What this means is that a change in the upwards z-axis direction represents a commensurate change in the temporal state of the agent.

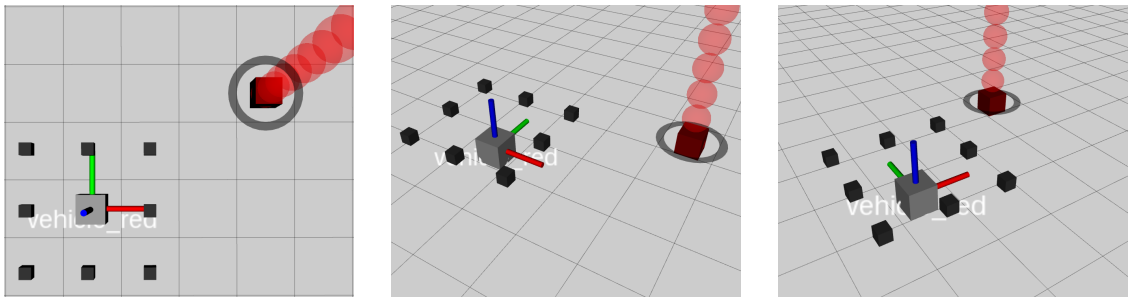
This way of visualising spatiotemporal information is illustrated in Figure 4.6, where the basic planning exercise is repeated with the spatiotemporal features of the planning included. As can be seen, the newly visited nodes now have a temporal component as well, as is indicated by them having a positive z component. The goal region has also extended upwards, indicating that it represents a range of spatiotemporal states, with the same spatial components. This makes sense, as the goal represents a point in space that must be reached, regardless of what time it is reached.

The new action space of the trajectory planner is as follows:

$$\begin{aligned}[\Delta x, \Delta y, \Delta t] \quad \in \quad \{ \; & [1,0,1], [1,1,1], [0,1,1], [-1,1,1], \\ & [-1,0,1], [-1,-1,1], [0,-1,1], [1,-1,1] \; \}\end{aligned} \tag{4.18}$$

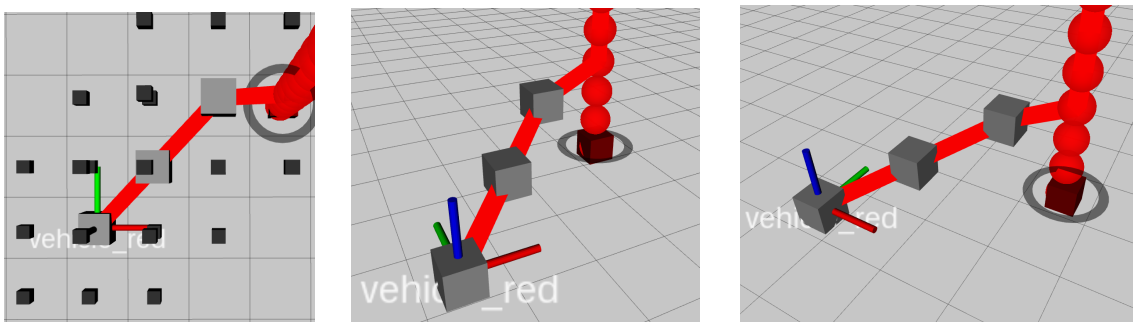
From this it can be observed that the action space has remained the same, except for the fact that each action now results in a temporal change of positive one. This indicates that executing an action results in the agent moving forward one unit in time.

The completed three dimensional path finding exercise can be seen in Figure 4.7, with the visited nodes being hidden in Figure 4.7(b) and (c) for visual clarity.



(a) **(b)** **(c)**

Figure 4.6: Visualising space-time by using the upwards z direction to represent temporal state.



(a) (b) (c)

Figure 4.7: An example of spatiotemporal planning.

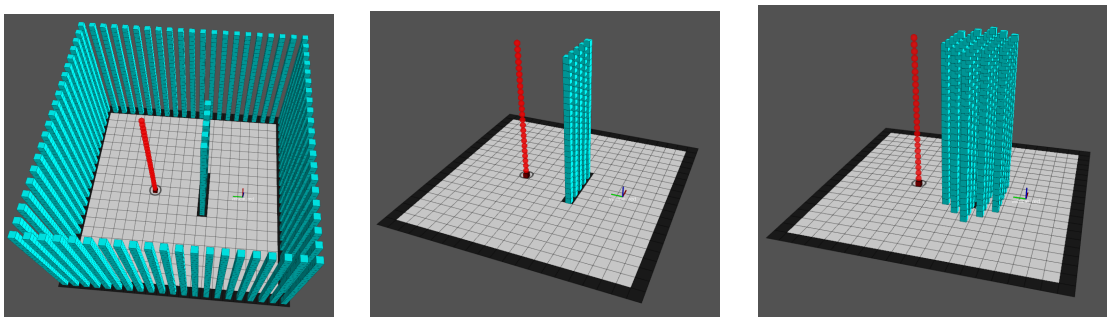
Space-time obstacles

As a result of planning in space-time, the free configuration space of the agent now has a temporal dimension. In order to find the free configuration space of the agent, it is first necessary to know what the occupancy space of the agent is, with the occupancy space being the space-time region that is occupied by the static obstacles in the environment as well as the trajectories of the other vehicles. This occupancy space has a temporal dimension, and can be visualised in a similar way as before, by using the spatial z-axis to represent the temporal state. This is illustrated in Figure 4.8, where the occupancy space is represented with the cyan cubes.

In Figure 4.8 (a), the entire occupancy space is shown, but the edges of the map can be visually obstructive, so they have been hidden in (b) and (c). Figure 4.8 (c) shows how the occupancy region would look if the footprint of the vehicle is taken into account. This is accomplished by inflating the occupancy space by the size of the vehicle's footprint.

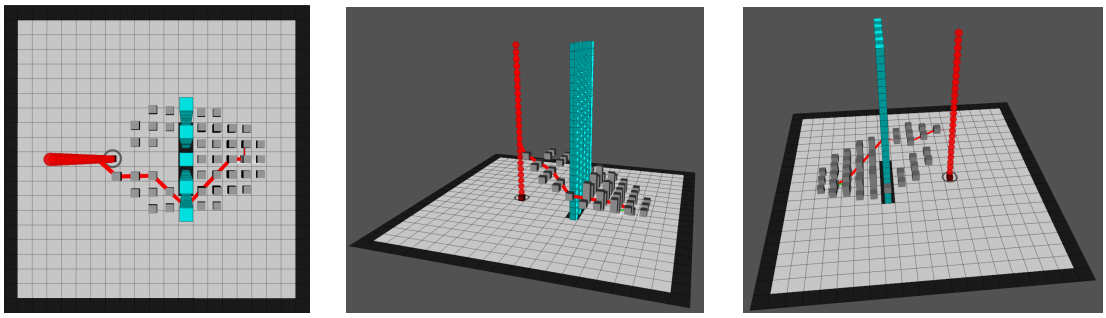
Using this way of representing space-time occupancy, the trajectory planning module can successfully find and return viable trajectories that the vehicle can execute. An example of finding a trajectory in the presence of space-time obstacles is shown in Figure 4.9.

This way of representing space-time obstacles also allows for visualising the trajectories of dynamic obstacles. Figure 4.10 shows this, with the dynamic obstacle having slightly



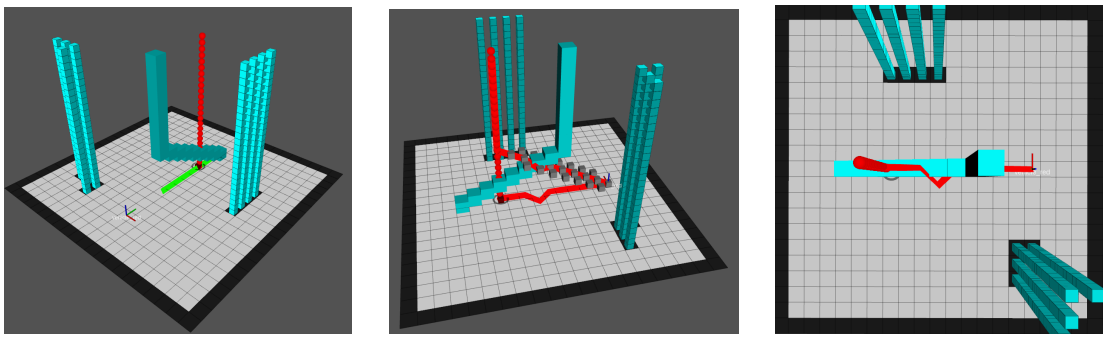
(a) (b) (c)

Figure 4.8: An example showing the representation used to visualise space-time occupancy, where the cyan cubes are the obstacles.



(a) (b) (c)

Figure 4.9: An example where a trajectory is found using the spatiotemporal occupancy space representation.



(a) (b) (c)

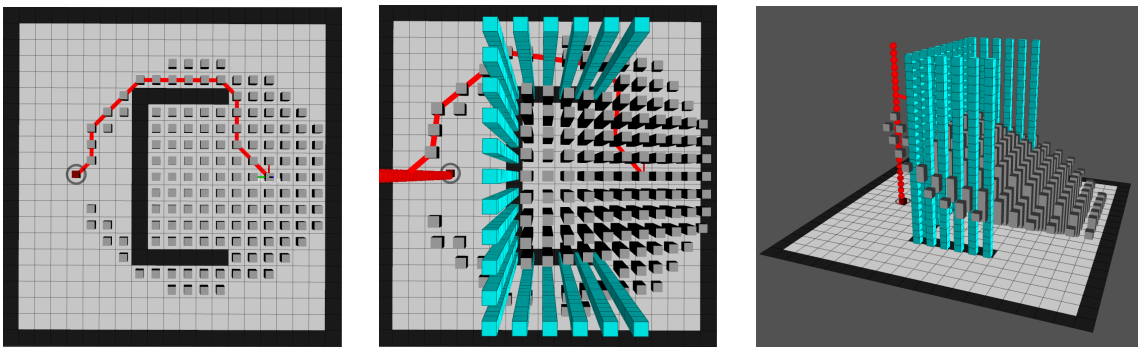
Figure 4.10: Finding a trajectory in space-time in the presence of dynamic obstacles. The green line is the path of the dynamic obstacle, and the vehicle's path and trajectory is shown using the red lines.

larger blue cubes. When using the z-axis for visualising the temporal changes, the z=0 plane represents the instant when the planning starts, and the planes that stack in the positive z direction represents the future state of the environment. In this example the dynamic obstacle moves toward the agent, and then stands still. The planner successfully finds a trajectory that avoids the dynamic obstacle and reaches the desired goal.

Using improved heuristic

One of the problems when planning in space-time is that by adding another dimension to the search space, the number of possible nodes that can be explored increases significantly. This is referred to as the "curse of dimensionality", and can cripple the usefulness of the search algorithm. As an example, consider Figure 4.11, where the map forms a cove that causes the search algorithm to "dam up" as it searches for a path to the goal. In the case of the path planner, where the temporal dimension is not used, the total number of explored nodes was 169. In contrast, when using the trajectory planner with the three-dimensional spatiotemporal state space, 785 nodes where explored before a valid trajectory was found.

One way of ameliorating this problem is to use a better heuristic. Most of the nodes that were explored in both cases were the ones inside the cove. If the search algorithm



(a) (b) (c)

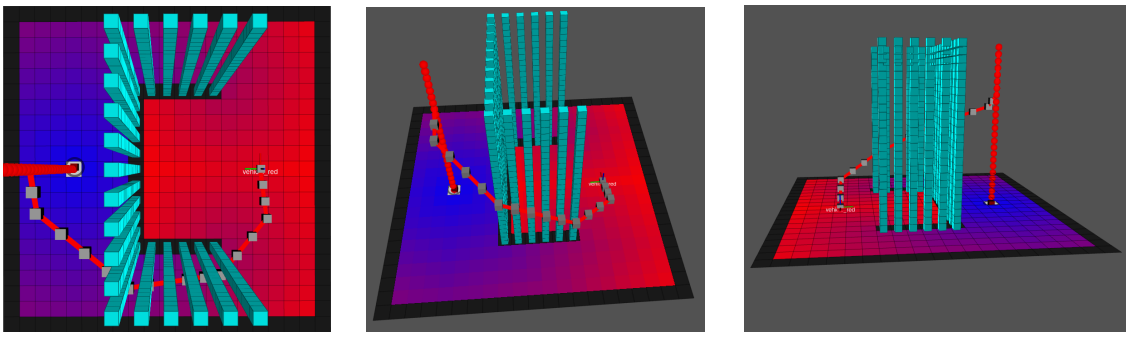
Figure 4.11: An example that illustrates the "curse of dimensionality" problem. In (a) the search is done in only two dimensions, whereas in (b) and (c) the search is done in three dimensional space-time.

somehow knew that there would be a dead end there, and could avoid searching the cove in the first place, then the number of explored nodes could be drastically decreased. This is illustrated in Figure 4.12, where only 18 nodes were explored before the path to the goal was found. The reason that this search was so effective was because a good heuristic was used that accurately represented the actual CTG, taking into account the effect of the static obstacles on the path cost. This heuristic is shown in Figure 4.12 as a colour-map, where the nodes that are close to the goal position are indicated using a blue colour, and the nodes that are far from the goal region are indicated using a red colour.

Using Reverse A* to obtain a good heuristic

The previous section illustrated the benefit of using a good heuristic to guide the order in which nodes are explored. This section will focus on the design of an algorithm that can be used to generate such a heuristic.

Silver (2005) describes an algorithm called Reverse Resumable A* (RRA*) which can be used to generate a good heuristic to guide the cooperative planning algorithm. The



(a) (b) (c)

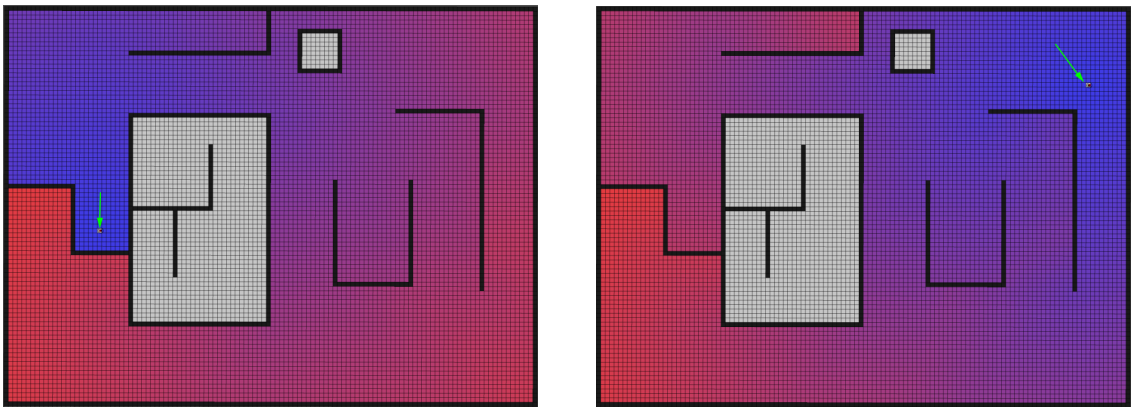
Figure 4.12: An example that illustrates the advantage of using an accurate heuristic to guide the search. Only the necessary nodes are explored to reach the desired goal location.

use of this heuristic is an example of hierarchical planning, where a search done in a lower-dimensional abstract space is used to guide the higher-dimensional search.

The RRA* algorithm is used to perform a lower-dimensional search in the spatial domain to generate a lookup table of the CTG from each node to the goal node, taking into account the static obstacles in the environment, but ignoring the dynamic obstacles. This lookup table is then used by the higher-dimensional spatio-temporal search algorithm as the heuristic function to provide a more accurate estimate of the CTG, and to guide the search for the optimal collision-free path, taking into account both the static obstacles and the dynamic obstacles.

One of the features of the RRA* algorithm is that it is resumable, meaning that it can be stopped and started as needed. This is useful when exploring a large environment, as it is not necessary to build a complete lookup table of the entire environment if only a smaller section of the environment needs to be searched. The resumable aspect of the RRA* was not used for this project and was therefore not implemented in the planning module. The complete heuristic lookup table was simply generated for the entire map before the performing the spatio-temporal planning. This approach worked well for the smaller environments that were used as the test cases for the simulation tests and the practical tests performed for this project. However, for the trajectory planner to scale to larger environments in future projects, the resumable aspect of the RRA* algorithm should also be implemented.

The heuristic lookup table is generated by performing a backward search in the spatial domain starting from the goal node and terminating when all reachable nodes in the environment have been visited. As the backward search executes, the CTC for each visited node is stored in a table, with the indices of the cells in the table corresponding to the positions of the nodes in the environment. Since the same node may be reached by multiple different paths from the goal node, the lowest CTC is stored in the table. Each time a node is visited, the new CTC is compared to the previous CTC which was stored in the table. If the new CTC is lower, then the CTC value in the table is updated, otherwise the new CTC is discarded. The CTC values in the table are all initialised with the same very large number, to ensure that the first time a node is visited, the new CTC will be lower than the initialised value stored in the table, and the table will therefore be updated using the "first visit" CTC value. The CTC values recorded in the table can also be used to determine whether a node has been visited yet. If the table entry corresponding to the node still contains the very large initial value, then it has not been visited yet. The search continues until all of the nodes in the environment have been reached, or until the open list is empty, and the remaining nodes are therefore unreachable. In the process, the CTC from the goal node to every reachable node in the environment is recorded in the table. The table of CTC values from the goal node to all of the reachable nodes now becomes the heuristic lookup table of the CTG from any node to the goal node. The usefulness of



(a) (b)

Figure 4.13: The resulting look-up-table after performing a reverse A* with respect to two different goal regions. The green arrows indicate the goal regions. Regions close to the goal location are represented using the blue colour, and red is used for regions far away from the goal location.

the reverse A* search is that it returns the true heuristic from any node, providing that there exists a valid path from that node to the goal node. It is important to note that the heuristic value is only the true heuristic insofar as the spatial domain is concerned, and does not take dynamic obstacles into account. The purpose of this heuristic is to minimise the time spent when searching for a valid trajectory in the spatiotemporal state space.

The cost function used when exploring the nodes during the path finding exercise can now use the lookup table generated by the reverse A*. This is done by changing the heuristic function $\hat{h}(n)$ in the following way:

$$\hat{h}(n) = H(i, j) \tag{4.19}$$

where H is the heuristic lookup table, and i and j are the x and y coordinates of the n^{th} node. The CTC was again calculated using the sum of the actions applied to reach the node, as follows:

$$\hat{g}(n) = \Sigma u_k \tag{4.20}$$

where u_k is the k^{th} action applied, and can be found using the following equation:

$$u_k = \sqrt{\Delta x_k^2 + \Delta y_k^2} \tag{4.21}$$

The total cost was calculated using an equally weighted sum of the CTC and CTG, as follows:

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n) \tag{4.22}$$

where $\hat{f}(n)$ represents the estimated total cost, and $\hat{g}(n)$ and $\hat{h}(n)$ represents the estimated CTC and CTG respectively.

The reverse A* heuristic lookup table generator also works for larger maps, as is

shown in Figure 4.13, where the green arrows indicate the goal positions and blue regions represents areas closer to the goal location. The effect of the obstacles are clearly visible, where some nodes that are close to the goal region have large CTG values due to being blocked off by obstacles. The grey areas in the map represent nodes that are unreachable from the goal node due to being completely blocked off by obstacles.

Adhering to kinematic constraints using manoeuvres

Up to this point, the planning has neglected to take into account any kinematic constraints that the vehicle might have, working under the assumption that the vehicle can translate in any direction with equal ease. This is unfortunately not always the case, as most vehicles have kinematic constraints that must be taken into account when executing trajectories. Some of the different kinematic constraints are discussed in Chapter 3, with the most relevant one being the unicycle dynamics, as that is the one that applies to the vehicles used in this project.

Unicycle dynamics, also known as differential drive dynamics, means that the position of the vehicle can be changed by using a combination of linear and angular velocities. Unlike bicycle dynamics, the unicycle model allows for rotating the orientation of the vehicle on one spot. This significantly simplifies the planning, as the turning circle of the vehicle does not need to be taken into account. It does, however, add more constraints than what has been used thus far. Until now, the planner has allowed the vehicle to translate in any of the intercardinal directions, regardless of the vehicle's orientation.

As a consequence of using the unicycle model, the vehicle's movement is constrained according to the direction that it is facing. In order for the planner to accommodate this, the orientation of the vehicle must be taken into account when planning. The mechanism that was used to accomplish this is the use of manoeuvres. The purpose of the manoeuvre is to abstract the kinematic constraints of the agent. In this way the action space of the agent can remain the same, and then the manoeuvres translate the action into a executable sequence of sub-actions.

When thinking about manoeuvres, it is important to distinguish between the world coordinate system, and the body coordinate system of the vehicle. This is necessary, because the action space of the vehicle is defined in terms of the world coordinates. For example, consider the scenario in Figure 4.14, where the differences between the world coordinate system and body coordinate system of the vehicle are illustrated. The different axes of the coordinate system are colour-coded according to the following mapping: [red, green, blue] = $[x, y, z]$. The heading of the vehicle is in the x-direction of its body axes system. In this example, the x-axis of the body coordinate system of the vehicle is pointing at a 135° angle with respect to the world x-axis. This means that one unit in the positive x-direction of the body axes does not correspond to one unit in the positive x-direction of the world coordinate system.

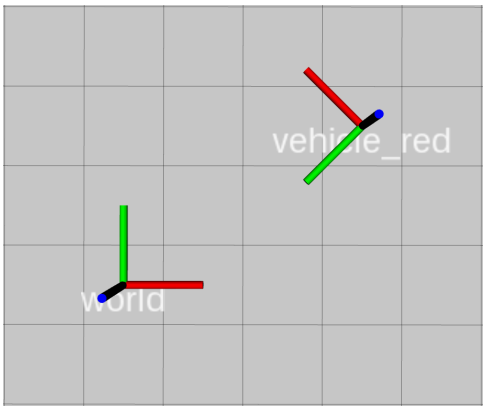


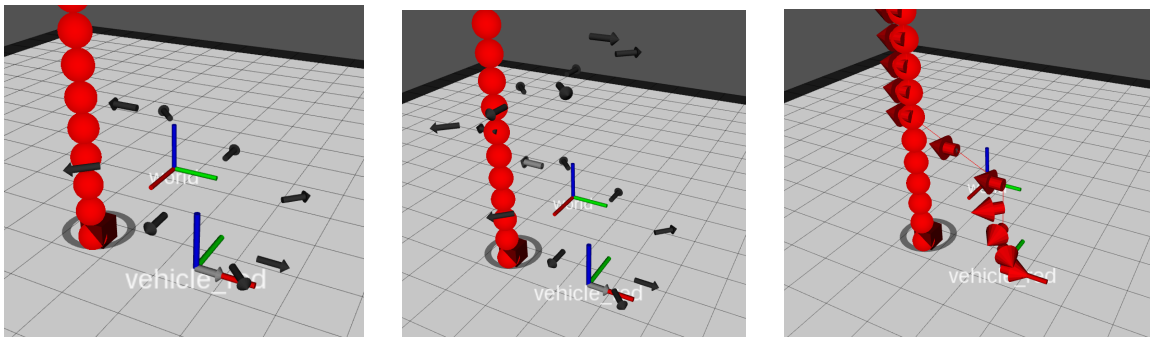
Figure 4.14: An example showing how the world coordinate frame and the vehicle body coordinate frame can differ.

When the action $[\Delta x, \Delta y] = [0, -1]$ is applied the scenario in Figure 4.14, then the position of the vehicle changes by negative one unit in the world y-direction. The vehicle is not facing in the y-axis direction though, so cannot translate in that direction. It would first have to rotate so as to be oriented in the y-direction, and then move one forward according to the body axis of the vehicle. Performing this rotation takes time, and must be taken into account during the planning phase. In order to accommodate the kinematic constraints of the vehicle, a manoeuvre planner is used to translate an action into a set of sub-actions that takes the orientation of the vehicle into account.

The manoeuvre planner is used when a new action is applied. The first step is to decompose that global action into a set of sub-actions that can be applied to the vehicle. The only admissible sub-actions that can be executed by the vehicle is to either rotate in the clockwise or anti-clockwise directions, or to move forward in the direction that it is facing. Therefore the first step is to rotate the vehicle so that it is facing in the direction that action is moving in. For example, if the required action is $[\Delta x, \Delta y] = [0, -1]$, then the vehicle has to rotate so that is facing in the y direction, having a heading of $-90°$. Only once the vehicle is facing in that direction can it move forward.

The different possible headings of the vehicle were chosen as all of the cardinal and intercardinal headings. One rotation in the clockwise or anti-clockwise directions changes the orientation of the vehicle to the next possible heading. For example, if the heading of the vehicle is $45°$ degrees, then one clockwise rotation would give the vehicle a heading of $0°$.

An example of where this manoeuvre-based planning is used to find a trajectory is shown in Figure 4.15. In Figure 4.15 (a), the neighbouring nodes are visited, which have been marked with the black arrows. The nodes that are not aligned with the direction of the vehicle have a larger temporal change, indicating the time needed to first perform the necessary rotation. Furthermore, the less a node is aligned with the direction of the vehicle, the more time is needed to perform the necessary rotations. The direction the



(a) (b) (c)

Figure 4.15: A manoeuvre-based planning approach. The actions that require more rotation have a larger temporal change component.

vehicle is facing is along the x-axis according to its body axes coordinate frame.

In Figure 4.15 (b), the node with the lowest total cost is explored, indicated by a dark grey arrow. From this point the next neighbouring nodes are visited. This continues until a suitable trajectory is found, which is shown in Figure 4.15 (c). The trajectory now includes the orientation of the vehicle, and how that orientation first changes before moving forward. The trajectory in this example can be expressed as follows: rotate in the clockwise direction four times, and then move forward two units.

When checking whether a node is admissible, the planner has to take into account the entire manoeuvre. For example, in Figure 4.15 the first action of the vehicle requires four rotations and one forward translation, totalling five moves. The space-time occupied by the vehicle while executing this manoeuvre will be five units, and all five those units have to be checked to see if they are available before the move can be allowed.

Choosing levels of discretisation

An aspect that has been disregarded until now is the discretisation resolution of the spatiotemporal state space. So far we have dealt with normalised units of movement, but have not translated them into real-world units. Ideally, the units used by the planner should map to units of measurement in the real world, such as meters or seconds. Choosing the appropriate resolution at which to discretise the environment is a trade-off. If the resolution is too fine, then storing the discretised version of the environment might be infeasible. On the other hand, if the resolution is too coarse, then potential solutions could be disregarded.

The resolution chosen for this project is that each spatial dimension has a resolution of 0.02 m, or 2 cm, and the temporal dimension has a resolution of 0.2 seconds. This means that if the agent moves 3 units in the x-direction in 5 units of time, then it would have moved 0.06 m in the x-direction in the real world over the course of 1 second. This resolution was chosen because it allows a sufficient level of detail to model the environment of the agent accurately, without the state space becoming prohibitively large. The total

number of nodes that are needed to model an environment can be calculated as follow:

$$N_{\text{nodes}} = \frac{l_x}{x_{\text{res}}} \times \frac{l_y}{y_{\text{res}}} \times \frac{l_t}{t_{\text{res}}} \tag{4.23}$$

For a map that is 2x2 meters and planning 10 seconds into the future, that would result in a state space of 500 000 nodes, as seen below

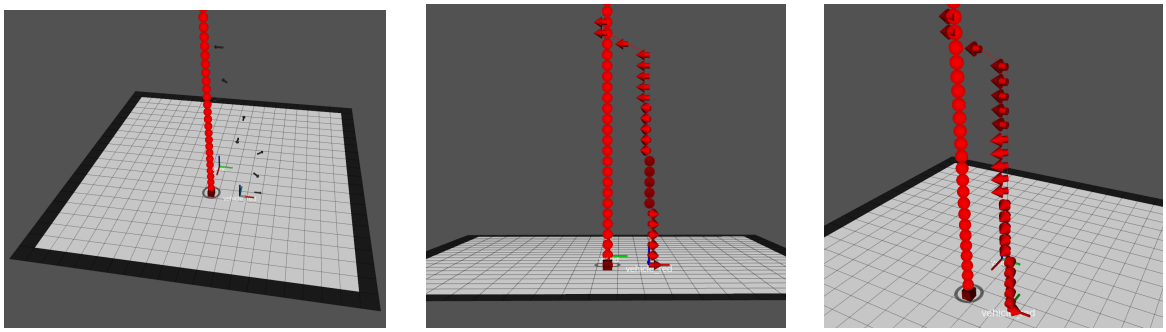
$$N_{\text{nodes}} = \frac{2}{0.02} \times \frac{2}{0.02} \times \frac{10}{0.2} = 500\ 000 \tag{4.24}$$

Appropriate temporal scaling for manoeuvres

One of the challenges that arose when choosing the appropriate level of discretisation was that the time taken to perform one forward translation does not necessarily correspond with the time taken to perform one rotation. For the level of discretisation chosen, the time allocated for a forward translation is 0.2 seconds, and the length of a forward translation is 0.02 m. This results in a linear speed of 0.1 m/s, which is entirely feasible for the vehicles used. However, when considering the angular speed, this is not the case. Performing one 45° rotation in 0.2 seconds results in an angular speed of 225 °/s, which is not executable by the chosen vehicles.

As a way of mitigating this problem, a ratio was used which described the number of time units needed to perform a rotation relative to one forward translation. A value of five was used for this ratio, which would allow the vehicle 5 time units, corresponding to 1 second, to perform one 45° rotation.

An example of where this ratio was used during planning is shown in Figure 4.16. In Figure 4.16 (a), the first nodes are explored, as is shown by the black arrows. The visited nodes have different temporal values, as is indicated by their height. The complete trajectory is shown in Figure 4.16 (b) and Figure 4.16 (c), with most of the time spent on performing a 180° rotation.



(a) (b) (c)

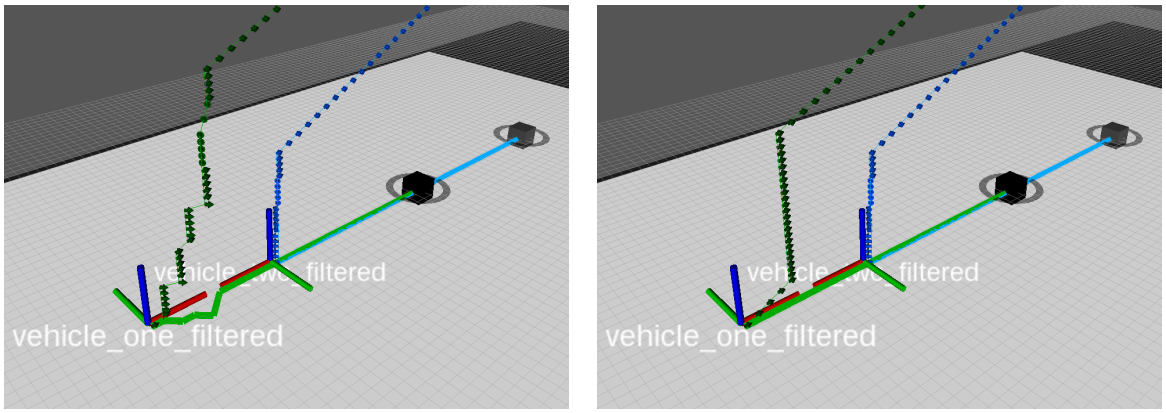
Figure 4.16: Manoeuvre-based planning with appropriate temporal scaling.

Allowing the vehicle to wait

When planning in a multi-agent environment, there are often situations where the desired trajectory of a vehicle is obstructed by the reserved trajectory of another vehicle. In these situations, a simple solution is for the vehicle to incorporate the ability to wait as part of its action space. This would allow the vehicle to remain stationary until the other vehicle has passed, and then to continue with its trajectory towards the goal region.

Up to this point the ability to wait has not yet been added. This results in strange behaviour, as seen in Figure 4.17 (a), where the vehicle performs a series of seemingly unnecessary rotation manoeuvres. In this example, there are two vehicles, as indicated by the two coordinate axes systems. Vehicle Two plans first, and finds a trajectory to the grey marker. The resulting path and trajectory are indicated with the blue colour. Vehicle Two is facing away from the goal marker, as indicated by the direction of the red axis, and therefore had to perform a 180° rotation first, which takes time. When Vehicle One plans, it has to take the reserved trajectory of Vehicle Two into account, including the time taken by its rotation. The optimal trajectory is for Vehicle One to wait for Vehicle Two to start moving towards its goal, and then to move along behind it towards its own goal.

In Figure 4.17 (a), the planner does not yet have the ability to stand still and wait, and so the trajectory included the rotations, as a means of forcing the vehicle to wait. As the rotations take time, this essentially allows the vehicle to wait without remaining stationary. If the ability to remain stationary is included in the vehicle's action space, the resulting trajectory is more optimal, and can be seen in Figure 4.17 (b). This is the expected behaviour, with Vehicle One waiting for Vehicle Two to start moving, and then following behind it.



(a) **(b)**

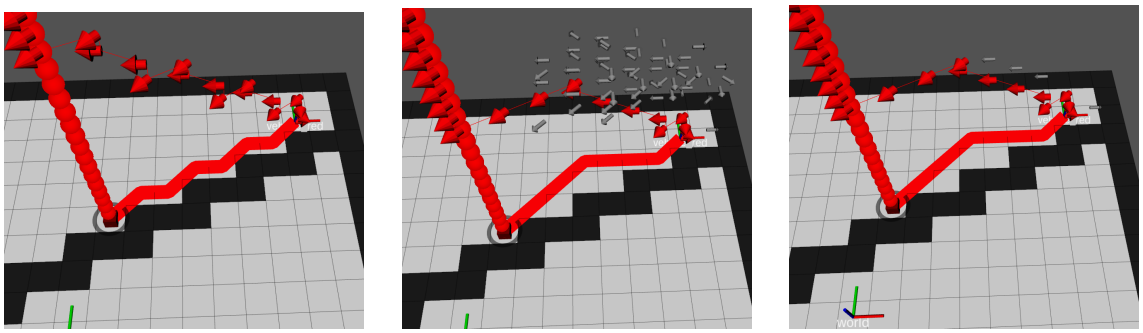
Figure 4.17: When planning around another vehicle's reserved trajectory, the best action is often for the vehicle to remain stationary for a time. In (a), the ability to remain stationary is not included in the action space, and the vehicle performs rotations to induce a delay. In (b), the ability to remain stationary is included in the action space of the vehicle, and the resultant trajectory is more optimal.

Minimising the number of rotations

As previously mentioned, the vehicles used for this project have unicycle kinematic constraints. This means that if they want to move in a direction other than the one they are facing, they need to rotate first. This rotation takes time, and should therefore be discouraged in order to find a more efficient path. One way to discourage rotate actions is to incorporate the cost of the rotate action into the CTC when visiting a new node. Up to this point the CTC has only been a function of the total distance travelled, irrespective of any rotations along the way.

Figure 4.18 shows how the trajectory that is returned by the algorithm changes when the CTC incorporates the cost associated with rotate actions. In Figure 4.18 (a), there is no cost associated with rotate actions, and the trajectory has more rotate actions than is necessary, resulting in a temporally longer trajectory. When the rotate actions are assigned an action cost, the trajectory exhibits only the minimal number of rotations as is seen in Figure 4.18 (b) and Figure 4.18 (c). As a consequence of including the rotation cost in the CTC, more nodes are explored than are necessary. This is because the CTG does not include the rotation cost, so it is always more expensive visiting a node than expected. This problem can be addressed by increasing the weighting of the CTG when calculating the total cost. Using a weighting multiplier of two for the CTG when calculating the total cost results in significantly less nodes being explored, as can be seen in Figure 4.18 (c).

The number of turns, total path length and number of explored nodes are shown for each scenario in Table 4.3. By reducing the number of turns from nine to five, the total path length is reduced from 17 units to 13 units. Additionally, when inflating the CTG heuristic, the number of explored nodes decrease from 68 to 12.



(a) **(b)** **(c)**

Figure 4.18: A comparison showing the effect that minimising the total number of rotations has on the path length can be seen in (a) and (b). When inflating the CTG heuristic, the effect on the number of explored nodes is seen in (c). The explored nodes are indicated by the grey arrows.

Table 4.3: The quantitative results from Figure 4.18

Figure number	Number of turns	Total path length	Number of explored nodes
a	9	17	8
b	5	13	68
c	5	13	12

Comprehensive example

Now that the design of the trajectory planner is complete it is possible to perform a comprehensive example to demonstrate all of its various aspects. This comprehensive example is presented in Figures 4.19 and 4.20, and shows how one of the vehicles can find a trajectory in the presence of static obstacles and another cooperative vehicle. The two vehicles are referred to as Vehicle Red and Vehicle Green, as indicated by the colours used. In Figure 4.19 (a) the pose of two vehicles are shown, as well as their respective goal positions as indicated by the flags. Figure 4.19 (b) shows the same scenario, but now the static environment has been discretised into 2 cm by 2 cm cells. In this scenario, Vehicle

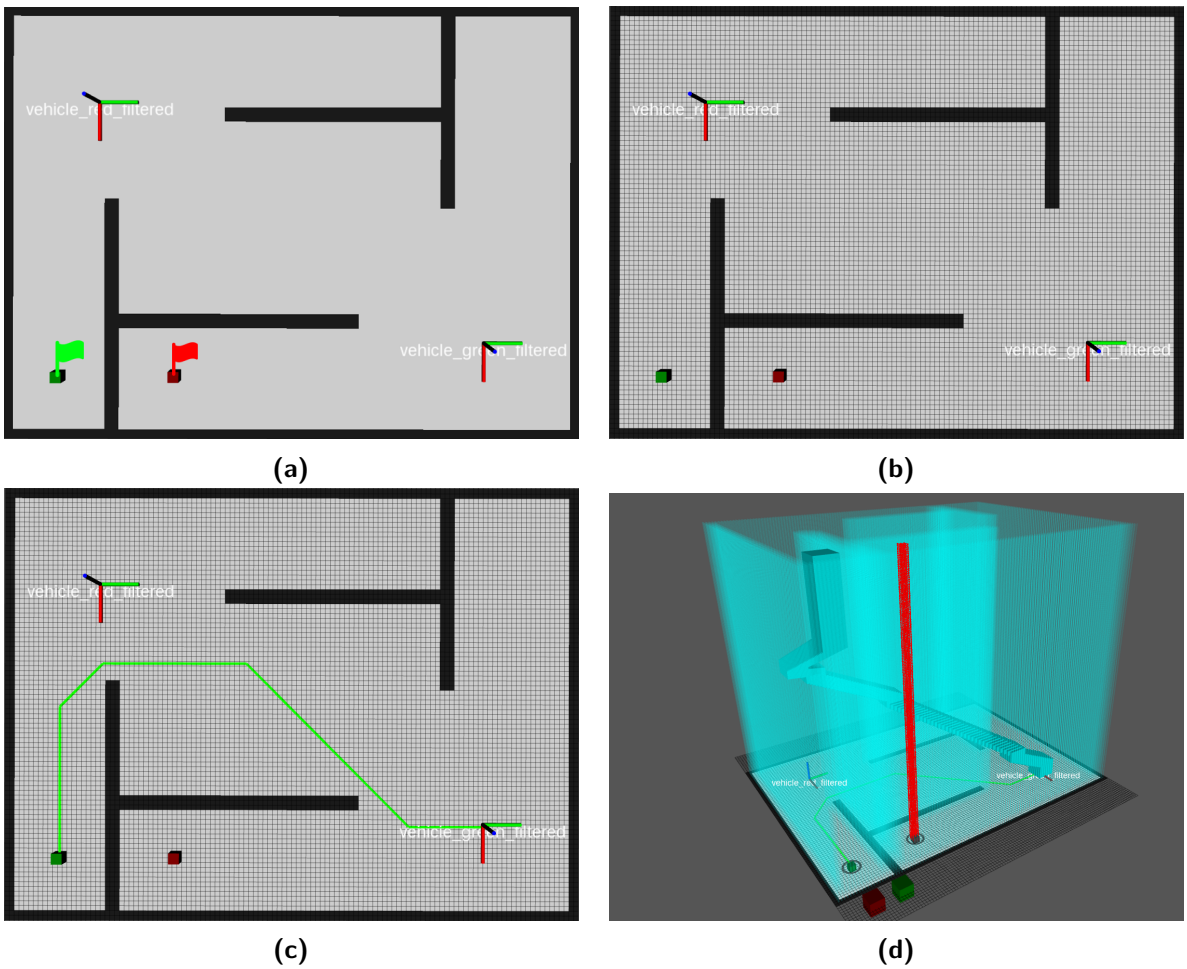


Figure 4.19: Part one of the comprehensive example showing trajectory planning in the presence of static obstacles and another cooperative vehicle.

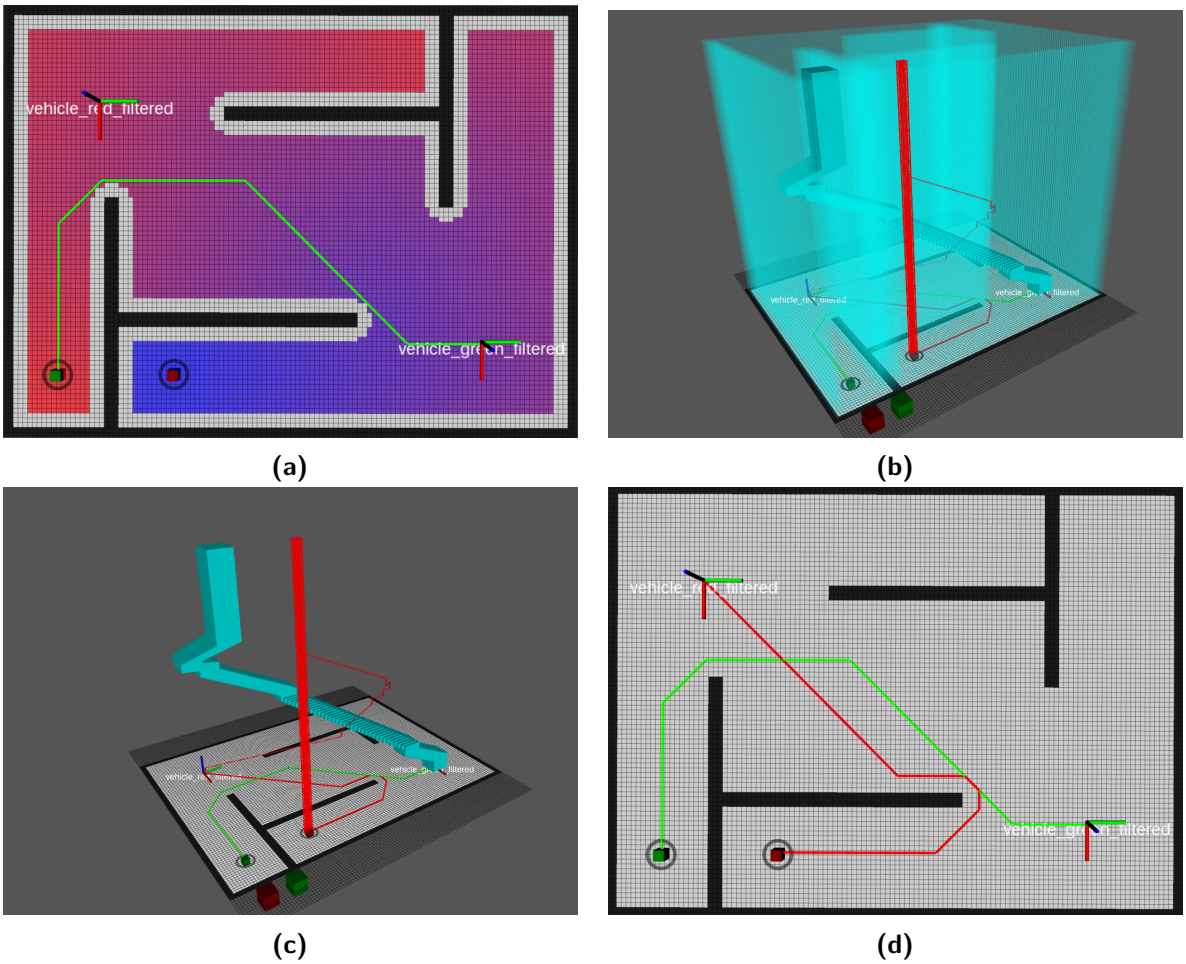


Figure 4.20: Part two of the comprehensive example showing trajectory planning in the presence of static obstacles and another cooperative vehicle.

Green has already found a trajectory, and has communicated that trajectory to Vehicle Red. This trajectory is shown in Figure 4.19 (c). Using the map of the static environment as well as the information contained in the trajectory communicated by Vehicle Green, Vehicle Red is able to construct the spatiotemporal occupancy space of the environment, as shown in Figure 4.19 (d). For this example, the static occupancy map is inflated with a vehicle footprint of 2 units. The trajectory of the other vehicle is inflated by the sum of the two vehicles' footprints, a total of four units. Figure 4.19 (d) also shows the goal region that Vehicle Red must reach, as indicated by the red column.

The first step in the trajectory planning is to find the heuristic lookup table using the reverse A* algorithm, the results of which is shown in Figure 4.20 (a), where the blue colour represent regions that are close to the goal area, and the red colour represents regions that are far from the goal area. Once the heuristic lookup table has been found, it is possible to find the trajectory to the goal position, as shown in Figure 4.20 (b). This same trajectory is shown in Figure 4.20 (c), but the static obstacles have been excluded from the occupancy space for the sake of visual clarity. Figure 4.20 (d) shows the final trajectories of Vehicle Red and Vehicle Green from a top-down perspective.

4.2. Coordination

The second component of the cooperative trajectory planner is the coordination framework that allows for decentralised multi-agent planning. This framework consist of three subcomponents, which are discussed in the following sections.

4.2.1. Reservation table

Cooperative planning relies on the vehicles sharing their intended trajectories with each other, and then planning around the trajectories of the other vehicles. For this to be possible, the other vehicles first have to communicate their trajectories, which then have to be stored in some way to be used when planning.

Silver (2005) presents a mechanism that can be used to keep track of the other vehicles' trajectories, calling it a reservation table. When a vehicle successfully finds a trajectory, it communicates that trajectory to all the other vehicles. This trajectory is then stored in the reservation tables of all the other vehicles. When the other vehicles plan, they can then treat the trajectories in their respective reservation tables as dynamic obstacles, and plan around them. Using this simple mechanism, it is possible for vehicles to plan in a cooperative fashion.

4.2.2. Token allocation

The immediate problem that presents itself when using the reservation tables, is that no two vehicles can plan at the same time. This is because for the vehicles to plan around each other, they need to take one another's trajectories into account. This is not possible if more than one vehicles plans at the same time, as they would not be able to take each other into account while planning.

The solution to this problem is to use a token allocation strategy along with a predefined priority for each vehicle. Consider a scenario where there are three vehicles, with preassigned priorities. Vehicle One has a priority level of one, the highest priority. Vehicle Two has priority level two, and Vehicle Three has priority level three, making Vehicle Three the lowest priority. Initially all three vehicles want to plan a trajectory, which they indicate by publishing their priority level on a communication channel that is shared between all the vehicles. After a vehicle has published its priority level it checks to see that no other vehicles with higher priorities have indicated they want to plan. If a vehicle with higher priority has also indicated that it wants to plan, then the vehicle with lower priority has to wait until the higher priority vehicle has finished planning. When a vehicle has started planning, it publishes its priority level again to notify the other vehicles. Once a vehicle has finished planning and has communicated its planned trajectory with the other vehicles, it publishes its priority level a third time to indicate that the next vehicle can start planning.

In this case, Vehicle One would start planning, as there are no other vehicles with higher priorities that want to plan. Once Vehicle Two sees Vehicle One publishing its priority a third time, it knows that it can start planning. This process repeats until all the vehicles that want to plan have done so.

An example of what the output of the priority sharing channel would look like in the above scenario is shown below:

timestep	: published content	*description*
t1	: 1,2,3	*All the vehicles indicate they want to plan by publishing their priority levels on the shared communication channel.*
t2	: 1	*Vehicle One sees that it has the highest priority, and so starts planning.*
t3	: 1	*Vehicle One has finished planning and indicates this by publishing its priority level.*
t4	: 2	*Vehicle Two sees that Vehicle One has finished planning, and starts planning its trajectory.*
t5	: 2	*Vehicle Two has finished planning and indicates this by publishing its priority level.*
t6	: 3	*Vehicle Three sees that Vehicle One and Vehicle Two have finished planning, and starts planning its trajectory.*
t7	: 3	*Vehicle Three has finished planning and indicates this by publishing its priority level.*
t8	:	*All the vehicle have successfully completed planning their trajectories.*

In order to understand why it is necessary for the vehicles to publish their priority level three times, consider the following scenario. Vehicle Two publishes its priority the first time, indicating that it wants to start planning. After is sees that no other vehicle wants to plan, it publishes its priority level a second time and starts planning. Now Vehicle One wants to start planning, and publishes its priority level. If Vehicle Two has only published its priority level once, and has not started planning yet, then Vehicle One can start planning seeing as it has the higher priority, and Vehicle Two would have to wait

for it to finish. However, if Vehicle Two has published its priority level twice, then it has started planning, and Vehicle One has to wait for it to finish before it can start planning. If this was not the case, then Vehicle One would start planning, and both Vehicle One and Vehicle Two would plan and publish their trajectories without taking each other into account. If Vehicle Three were to also indicate that it wanted to start planning while Vehicle Two was still busy, then the difference in priority between Vehicle One and Vehicle Three would decide who gets to plan first once Vehicle Two has finished.

4.2.3. Windowing

Windowing is another features that Silver (2005) uses to facilitate cooperative planning. Windowing allows the planning to be implemented in an online way by automatically replanning the trajectory of the vehicle at a fixed interval, and only publishing the trajectory of the vehicle for a fixed window into the future. The advantage that this feature offers is that it allows the planning module to respond to changes in the environment, such as the other vehicle's trajectories changing as a result of replanning. Consider the following example, where two vehicles plan in the same environment. Vehicle One plans a trajectory to its goal location, and communicates this to Vehicle Two. When Vehicle Two plans its trajectory, it takes the trajectory of Vehicle One into account. Unfortunately the trajectory reserved by Vehicle One results in Vehicle Two having to use a highly suboptimal trajectory, as the space-time that it would have wanted to use is already reserved by Vehicle One. Nevertheless, it uses the suboptimal trajectory seeing as it is the best that is available. Some time later, Vehicle One's destination changes, and it replans accordingly. This new trajectory would have allowed Vehicle Two to find a more optimal trajectory, as it occupies less of the space-time that Vehicle Two would also have wanted to use. Without windowing, Vehicle One would have continued using its original trajectory. By using windowing, each vehicle replans periodically, taking any changes in the other vehicles trajectories into account. This would enable Vehicle Two to find a more suitable trajectory, which takes Vehicle One's changed trajectory into account.

4.3. Implementation

The algorithm that has been incrementally designed in the previous sections can now be used to accomplish the cooperative trajectory planning. The overall architecture of the cooperative trajectory planning module is shown in Figure 4.21. The module subscribes to five ROS topics, namely the tokens topic, the odometry of the vehicle, the desired goal location, the trajectories of the other vehicles, and the map of the static environment. The "/tokens" topic is the shared communication channel where the vehicles publish their priority levels to coordinate their planning. The system architecture included

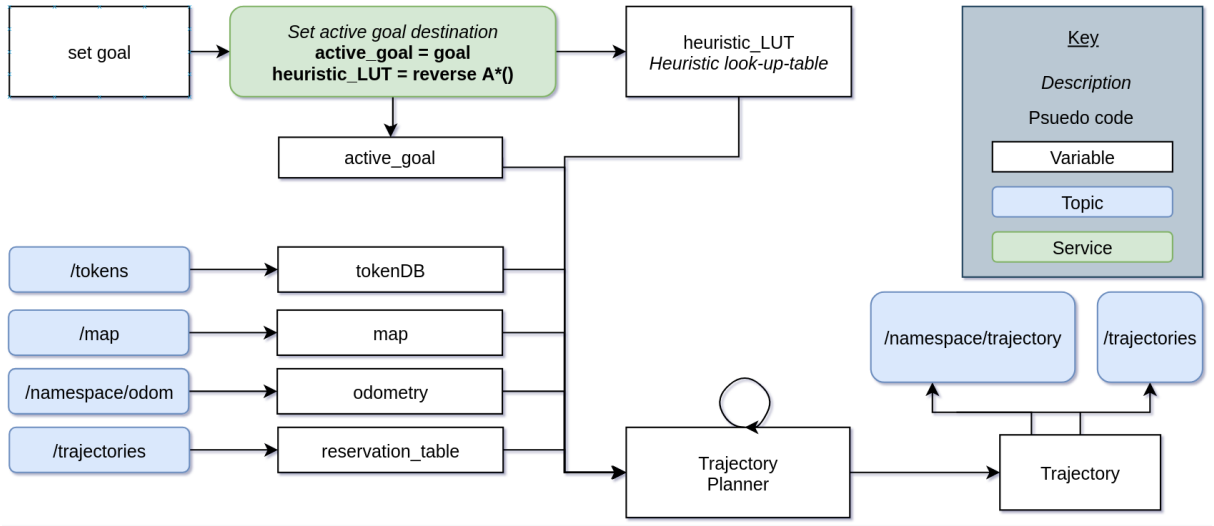


Figure 4.21: The complete cooperative trajectory planning module.

a ROS service, which is used to synchronously change the active goal location of the vehicle. The active goal is the one that is used for the trajectory planning. Lastly, the module publishes the planned trajectory to two different topics, the first of which is the "/<vehicle_namespace>/trajectory" topic, where the vehicle_namespace would be "vehicle_one", "vehicle_two", etc. This topic is used to communicate the reference trajectory to the vehicle's trajectory tracking module. The second topic is the "/trajectories" topic, which is used the communicate the trajectory to the other vehicles for updating their reservation tables.

4.4. Summary

This chapter presented the design of the cooperative trajectory planning module. This was done by first designing an algorithm that can find a trajectory from a starting pose to a goal location, taking the map of the static environment into account as well as the reserved trajectories of the other vehicles. Additionally, this algorithm is able to adhere to the kinematic constraints of the vehicle by using a manoeuvre-based planning approach. Secondly, a coordination framework was designed that could be used to facilitate the decentralised planning of the vehicles. This coordination framework makes use of a reservation table on each vehicle that can be used to record the reserved trajectories of the other vehicles. Furthermore, it uses a prioritised token allocation strategy when deciding which vehicle plans when, so that no two vehicles plan at the same time. Lastly, it implements a windowing technique which enables the trajectory planning to occur at a regular interval, allowing the vehicles to reserve a new trajectory should they deviate from their previous one.

Chapter 5

Trajectory Tracking

The purpose of the trajectory tracking module is to listen for and execute trajectories that are published by the cooperative trajectory planner. The trajectory tracking module uses a Model Predictive Control (MPC) approach to accurately track the planned trajectories, and consists of two components: the trajectory optimiser and the trajectory executer. This architecture can be seen in Figure 5.1. The trajectory optimiser component is responsible for finding the optimal set of velocity commands that the vehicle must execute to follow the planned trajectory. The trajectory optimisation is performed periodically and for a limited time window into the future. The trajectory execution component is responsible for sending the velocity commands that have been calculated by the trajectory optimisation component to the vehicle's velocity controller.

5.1. Trajectory optimisation

The purpose of the trajectory optimiser is to calculate the sequence of velocity commands for a given time window that can by executed by the vehicle within its dynamic constraints, which would enable the vehicle to follow the reference trajectory provided by the trajectory planner as closely as possible. The trajectory optimisation problem is formulated as a nonlinear programming problem, which can be solved using any of a number of different tools. For this project, the CasADi tool by Andersson *et al.* (2019) was selected because it is well documented and is capable of solving nonlinear optimisation problems. Internally, CasADi makes use of the IPOPT solver, with the Euler collocation integration method.

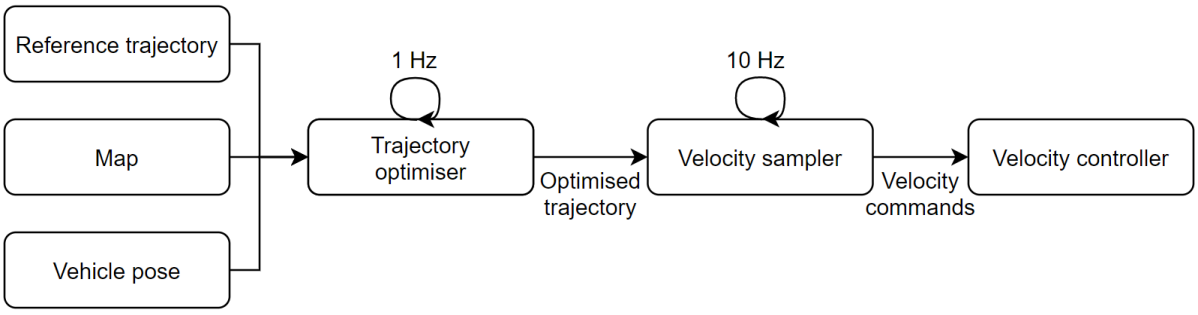


Figure 5.1: Architecture of the trajectory tracking module.

A nonlinear optimisation problem is formulated using an objective function and a set of constraints, which can be nonlinear in nature. The goal of the nonlinear optimisation tool is to minimise the objective function subject to the given constraints. One of the challenges with the optimisation is that the objective function can have multiple local minima that can cause the optimisation to return sub-optimal solution. To address the problem of local minima, as well as to reduce the number of iterations required to find the solution, the optimisation process is seeded with a good initial solution.

The optimisation problem is formulated mathematically as follows:

$$\begin{aligned} \underset{\bar{x}}{\text{minimise}} \quad & f(\bar{x}, p) \\ \text{subject to} \quad & \bar{x}_{lb} \leq \bar{x} \leq \bar{x}_{ub} \\ & g_{lb} \leq g(\bar{x}) \leq g_{ub} \end{aligned} \tag{5.1}$$

where $f(\bar{x}, p)$ is the objective function, $\bar{x}$ is the state trajectory of the vehicle, and $g(\bar{x})$ is the set of constraint functions. The state trajectory is the time sequence of the vehicle states, consisting of its position, velocity, heading and heading rate. To illustrate the development of the trajectory optimisation algorithm in this chapter, the state trajectory will initially only consist of the position and velocity, but will later be expanded to include the heading and heading rate as well. Both the state trajectory and the constraint function can be given upper and lower bounds, to decrease the size of the problem space. The objective function $f(\bar{x}, p)$ is expressed as the weighted sum of cost functions $f_i(\bar{x})$, where p_i is the weight allocated to each cost function.

$$f(\bar{x}, p) = \sum_{i=0}^{I} p_i f_i(\bar{x}) \tag{5.2}$$

For the purpose of this project, the optimisation is required to generate a sequence of velocity commands that can be executed by the vehicle so that it can adhere to a given set of space-time poses. These space-time poses are usually supplied by the cooperative trajectory planner. The first step is to determine which of these space-time poses are relevant to the optimisation exercise. This can be done by considering which poses lie in the time window between the current time when the initialisation is started and the horizon time that is specified beforehand. The optimiser also adds its current space-time pose to the list of poses. Once it has all the relevant poses, it interpolates along those poses according to a time discretisation that is set beforehand. This results in a trajectory that is similar to what the final version will look like, and is used as the starting seed for the optimisation process. Using this seed results in less iterations when the optimisation is performed.

5.1.1. Holonomic vehicle

The trajectory optimisation algorithm was developed incrementally, starting with the most basic functionality. More complexity was only added once the basic functionality worked. To this end, the nonholonomic constraints of the vehicle were ignored initially, as well as the obstacle clearance constraints. This meant that the vehicle was treated as a first-order linear model, with state vector $\bar{x}$ expressed as

$$\bar{x} = [x, y, \dot{x}, \dot{y}]^T \tag{5.3}$$

The objective function used for this version of the algorithm consisted of the weighted sum of two cost functions, shown in Equations (5.4) and (5.5). f_0 represents the control effort that the vehicle uses, which is proportional to the velocity of the vehicle, and is defined as follows:

$$f_0 = \sum_{j=0}^{J} \dot{x_j}^2 + \sum_{j=0}^{J} \dot{y_j}^2 \tag{5.4}$$

The cost function which measures the trajectory adherence of the vehicle is represented by f_1, and is defined by the following equation:

$$f_1 = \sum_{j=0}^{J} (x_j - x_j^{ref})^2 + \sum_{j=0}^{J} (y_j - y_j^{ref})^2 \tag{5.5}$$

where x_j^{ref} and y_j^{ref} denote the poses of the reference trajectory. Figure 5.2 illustrates the distinction between the reference trajectory and the trajectory which is being optimised.

The objective function f is then defined as the weighted sum of the control effort cost

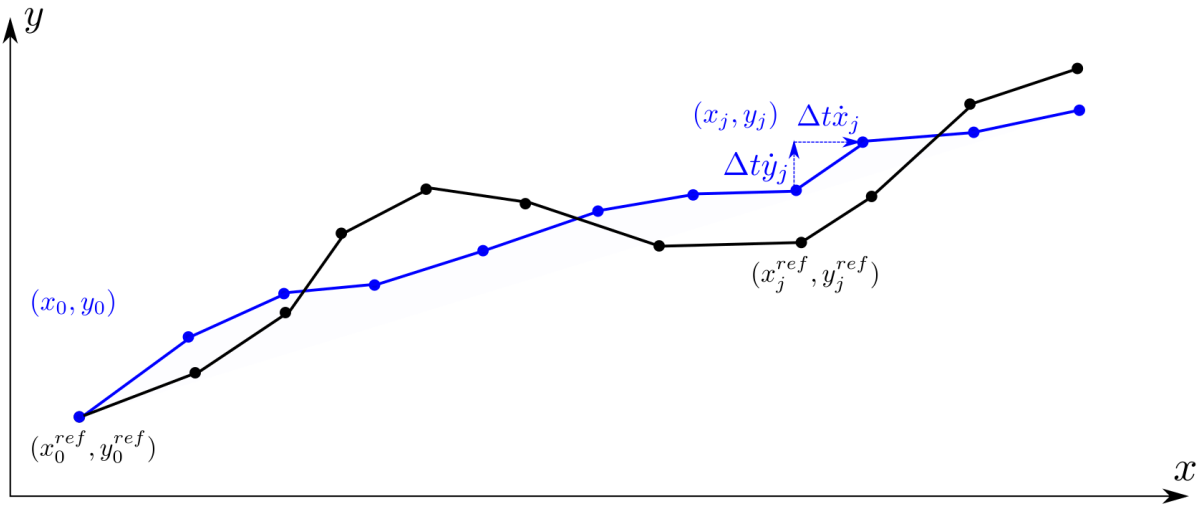


Figure 5.2: An example which shows the distinction between the reference trajectory and the trajectory that is being optimised, as well as the notation used when calculating f_0 and f_1. The reference trajectory is shown in black, while the trajectory being optimised is shown in blue.

function and the trajectory adherence cost function, as follows:

$$\begin{aligned} f(\bar{x}, p) &= p_0 f_0(\bar{x}) + p_1 f_1(\bar{x}) \\ &= p_0(\sum_{j=0}^{J} \dot{x}^2 + \sum_{j=0}^{J} \dot{y}^2) + p_1(\sum_{j=0}^{J}(x_j - x_j^{ref})^2 + \sum_{j=0}^{J}(y_j - y_j^{ref})^2) \end{aligned} \tag{5.6}$$

When performing the optimisation, the solver minimises the objective function subject to a list of constraints. The first constraint is that the optimised trajectory must accommodate holonomic kinematic constraints. This means that the vehicle is allowed to translate in any direction, regardless of its orientation, according to the following differential constraints:

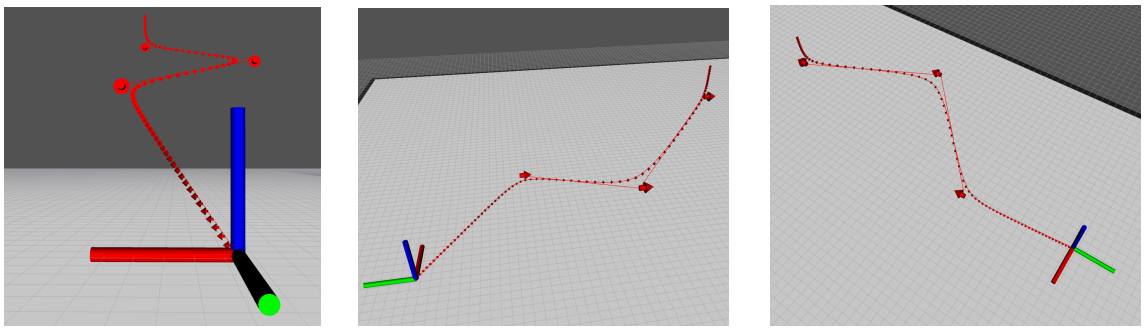
$$x_{j+1} = x_j + \Delta t \dot{x}_j, \forall j \tag{5.7}$$

$$y_{j+1} = y_j + \Delta t \dot{y}_j, \forall j \tag{5.8}$$

This holonomic constraint was later changed to a more restrictive nonholonomic constraint which allows only for motion according to the unicycle kinematics model, which better represents the vehicles that were used to test the algorithms in simulation and real life. The second constraint is that the first position in the optimised trajectory must be the same as the current position of the vehicle.

$$[x_0, y_0] = [x_0^{ref}, y_0^{ref}] \tag{5.9}$$

This forces the optimised trajectory to have its starting position the same as the current position of the vehicle, resulting in a smooth trajectory that is executable by the vehicle. Figure 5.3 shows an example of where this version of the optimisation module is used to find a trajectory, given a set of space-time poses. The space-time poses are indicated by the larger red arrows, while the optimised trajectory is indicated by the smaller red arrows. The space-time coordinates of the desired poses as well as the current pose of the vehicle are shown in Table 5.1. The position and orientation of the vehicle is represented



(a) (b) (c)

Figure 5.3: An example that shows the trajectory produced when the holonomic version of the trajectory optimisation module is used.

Table 5.1: The space-time poses which the optimiser had to adhere to.

Time	x [m]	y [m]	Heading [deg]
63.81	0.11	-0.83	180
70	0.00	-0.60	90
73	0.20	-0.40	90
76	0.0	-0.20	90

by the body axis system, where the red, green and blue axes point in the x, y and z body-axis directions respectively. The forwards direction of the vehicle is denoted by the red axis. Similar to the cooperative trajectory planner, the upwards z direction is used to represent the temporal direction. According to this, it can be seen from Figure 5.3 that the space-time poses of the state trajectory all have a temporal component, represented by their position along the z axis.

The holonomic nature of the optimised trajectory can be seen by how the vehicle starts translating along the trajectory without first changing its orientation. Given that the vehicle's forward direction is indicated by the red axis, it would have had to change its orientation first if it were nonholonomically constrained. The second constraint, which is that the trajectory needs to start at the current position of the vehicle, is also satisfied.

The reason for performing the optimisation is to calculate the velocities necessary for executing the desired trajectory. The velocity commands that were calculated for this trajectory can be seen in Figure 5.4. For this specific optimisation, a horizon of 15 seconds was used, with a time discretisation of 0.2 seconds. This resulted in an optimised trajectory that consisted of 75 space-time poses.

Note that the trajectory optimisation algorithm gives the commanded velocities in the world axis x- and y-directions, and not in the vehicle's body axis directions. The velocities are given in terms of the world axis x- and y-directions because the holonomic constraint imposed on the trajectory optimisation allows the vehicle to translate in any direction, independently from the direction that it is facing.

5.1.2. Nonholonomic vehicle with differential drive

Given that the optimisation works for the simplified case when only holonomic constraints are considered, the next step is to add more restrictive constraints to more accurately model the actual vehicles used when executing the trajectories. The vehicles used for this project have a differential drive mechanical system, which can be modelled by the unicycle kinematic model. The new state representation of the vehicle is as follows:

$$\bar{x} = [x, y, \theta, v, \omega]^T \tag{5.10}$$

where v and ω are the linear and rotational velocities of the vehicle. These velocities

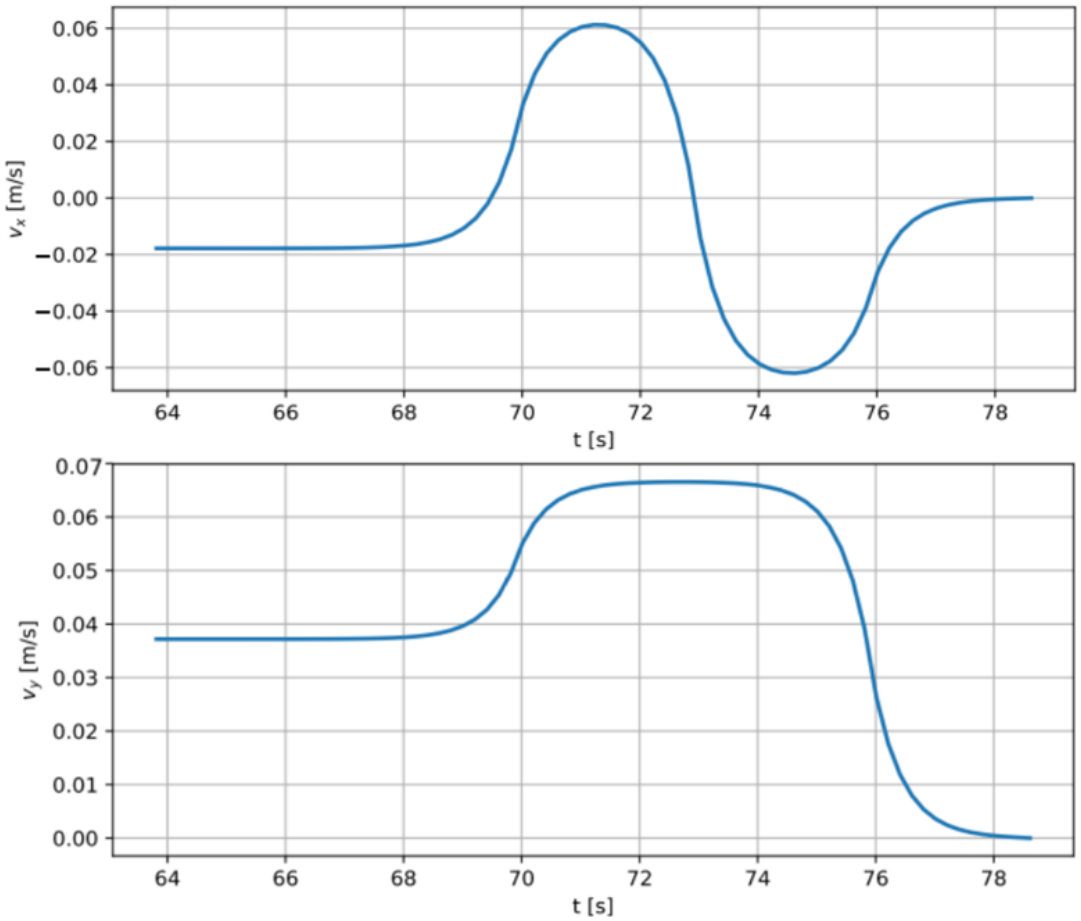


Figure 5.4: The velocity commands produced when optimising the trajectory using holonomic constraints. v_x and v_y are equivalent to $\dot{x}$ and $\dot{y}$ respectively.

are relative to the body axis of the vehicle, and can be executed by the vehicle's velocity controller.

The objective function remains mostly the same, with only the cost function f_0 changing to reflect the fact that the velocity of the vehicle now needs to be expressed in terms of the body axis of the vehicle.

$$f_0 = \sum_{j=0}^{J} v_j^2 + \sum_{j=0}^{J} \omega_j^2 \tag{5.11}$$

The objective function f can now be expressed in terms of the updated cost function

$$\begin{aligned} f(\bar{x}, p) &= p_0 f_0(\bar{x}) + p_1 f_1(\bar{x}) \\ &= p_0(\sum_{j=0}^{J} v_j^2 + \sum_{j=0}^{J} \omega_j^2) + p_1(\sum_{j=0}^{J}(x_j - x_j^{ref})^2 + \sum_{j=0}^{J}(y_j - y_j^{ref})^2) \end{aligned} \tag{5.12}$$

The differential constraints are also adjusted to model the nonholonomic constraints of the vehicle as follows:

$$x_{j+1} = x_j + \Delta t v_j \cos(\theta_j), \forall j \tag{5.13}$$

$$y_{j+1} = y_j + \Delta t v_j \sin(\theta_j), \forall j \tag{5.14}$$

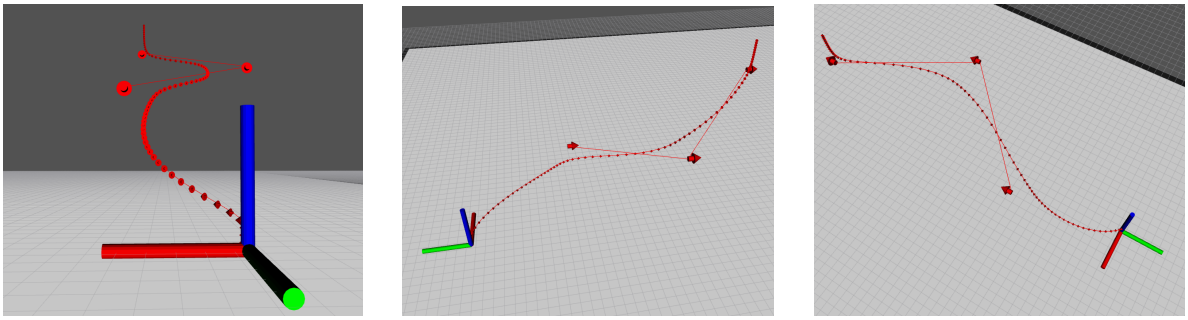
$$\theta_{j+1} = \theta_j + \Delta t \omega_j, \forall j \tag{5.15}$$

The starting position constraint is also changed to a starting pose constraint, as the orientation of the vehicle must be taken into account when optimising the trajectory as follows:

$$[x_0, y_0, \theta_0] = [x_0^{ref}, y_0^{ref}, \theta_0^{ref}] \tag{5.16}$$

When using the nonholonomic kinematic constraints during the optimisation process, the resulting trajectory is significantly different, as can be seen in Figure 5.5. The most conspicuous difference between this trajectory and the one produced when using only the holonomic constraints is that trajectory adherence has degraded. This is as a result of the fact that the optimiser now needs to allow time for the vehicle to change its orientation before translating in a direction. For example, at the start of the trajectory, the vehicle first rotates in one spot before moving forward, waiting until it is properly aligned with the direction in which it wants to translate. This behaviour can be more clearly seen in Figure 5.6, which shows the linear and rotational velocities produced by the optimiser. The linear velocity of the vehicle is zero for roughly the first half second, allowing enough time for the vehicle to rotate its orientation appropriately.

One way to mitigate the degrading of the trajectory adherence, is to increase its weighting in the objective function. This works because the objective function is the weighted sum of the effort used by the vehicle, and its trajectory adherence error. Increasing the weighting on the trajectory adherence part of the objective function will result in more effort being used by the vehicle to adhere to the trajectory. This increased effort manifests itself in increased velocities. Once the weighting has been changed, the resultant trajectory



(a) (b) (c)

Figure 5.5: An example that shows the trajectory produced when the nonholonomic version of the trajectory optimisation module is used.

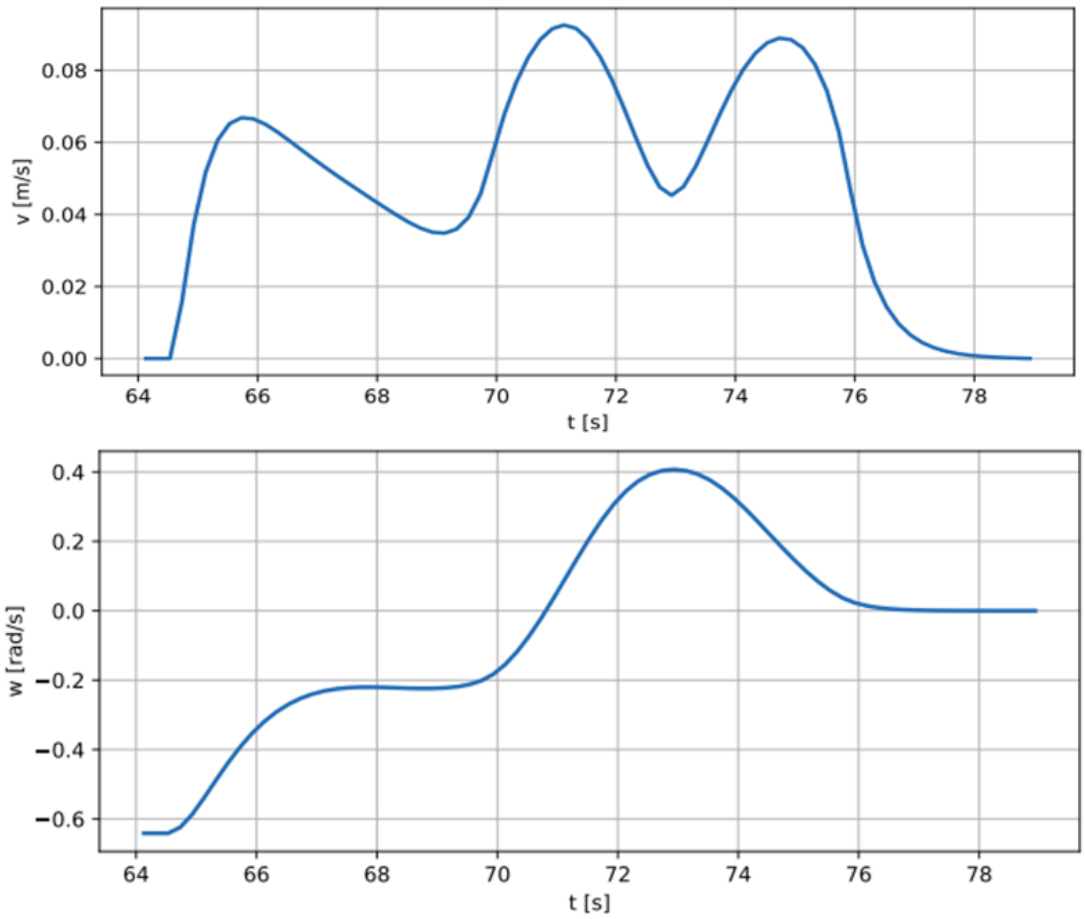
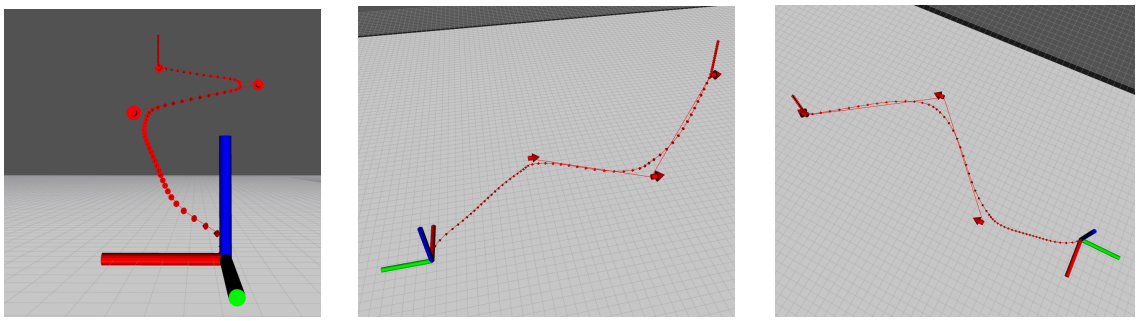


Figure 5.6: The velocity commands produced when optimising the trajectory using nonholonomic constraints.

does indeed adhere more closely to the planned trajectory, as can be seen in Figure 5.7 and Figure 5.8, with the velocities showing an increase as expected.



(a) (b) (c)

Figure 5.7: Another example that shows the trajectory produced when the nonholonomic version of the trajectory optimisation module is used. This time the weighting for the space-time adherence is increased.

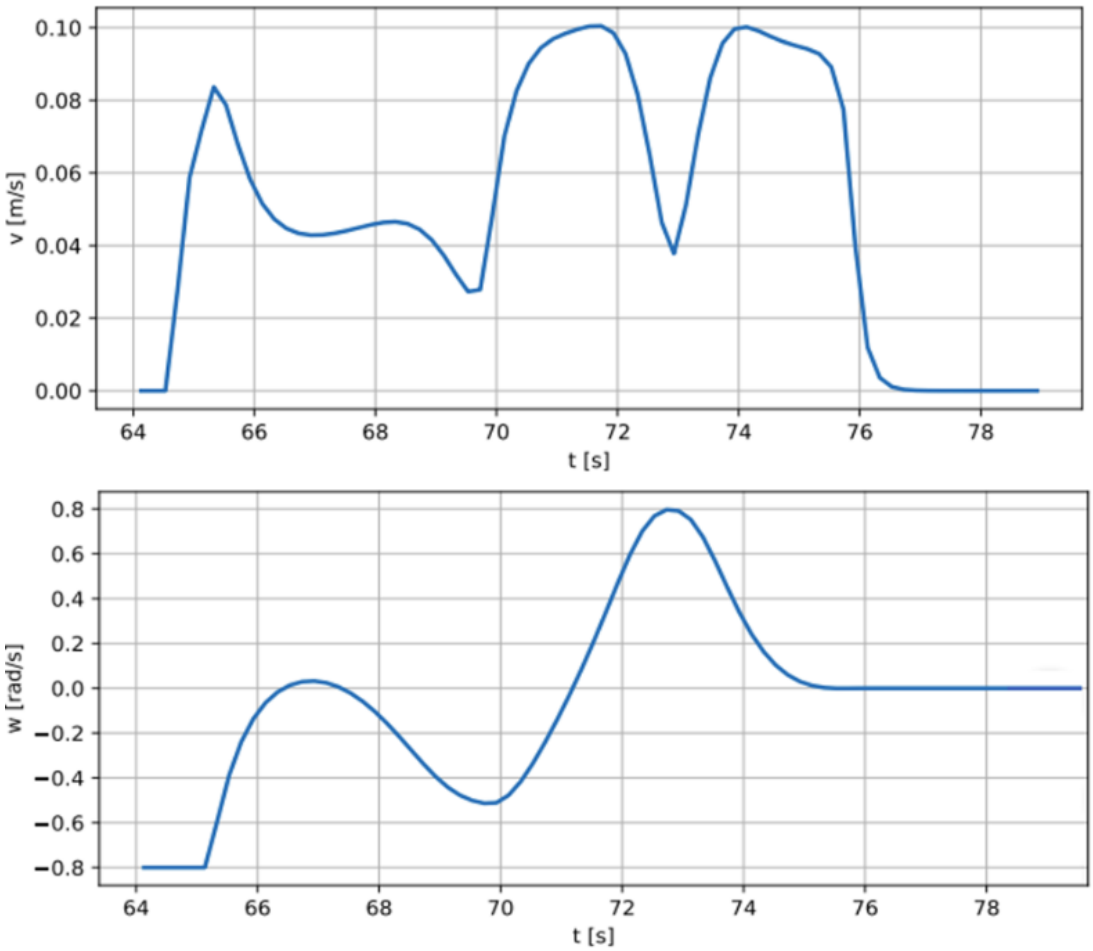
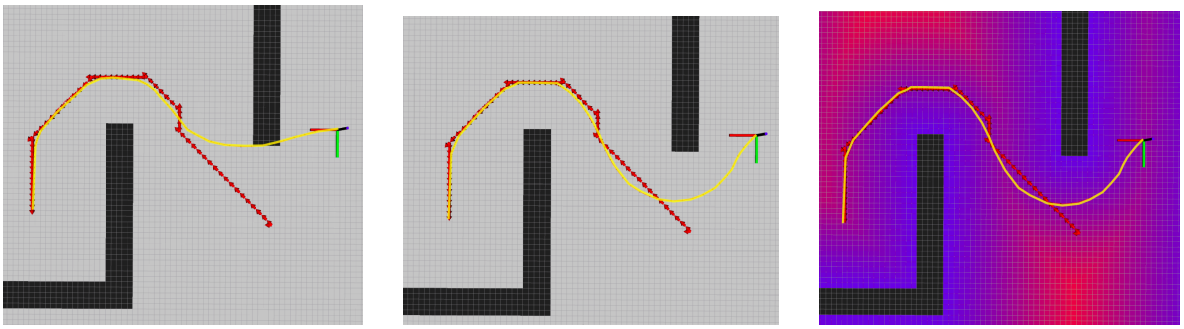


Figure 5.8: Another example of the velocity commands produced when optimising the trajectory using nonholonomic constraints, this time using an increased space-time adherence weighting.

5.1.3. Obstacle avoidance

The vehicle may stray from its planned trajectory due to model uncertainty or external disturbances. In this case, the trajectory optimiser must provide a trajectory that will allow the vehicle to rejoin the planned trajectory without colliding with an obstacle. Now that the optimiser is able to take the nonholonomic kinematic constraints of the vehicles into account, the final step is to add obstacle avoidance. The need to include obstacle avoidance in the trajectory optimisation is illustrated in Figure 5.9. In this scenario, the vehicle has strayed quite far from its planned trajectory. If the trajectory optimiser does not take the obstacles into account, it produces the trajectory shown in Figure 5.9 (a). If this trajectory were to be executed, it would result in the vehicle colliding with the static obstacle. If the trajectory optimiser takes the obstacles into account, it produces the trajectory shown in Figure 5.9 (b). In order to produce this trajectory, an additional constraint was added to the optimiser, one that takes the occupancy map of the vehicle's environment into account. This constraint uses a lookup table that contains the distance to the nearest obstacle from any point in the map. The lookup table is generated using an



(a) (b) (c)

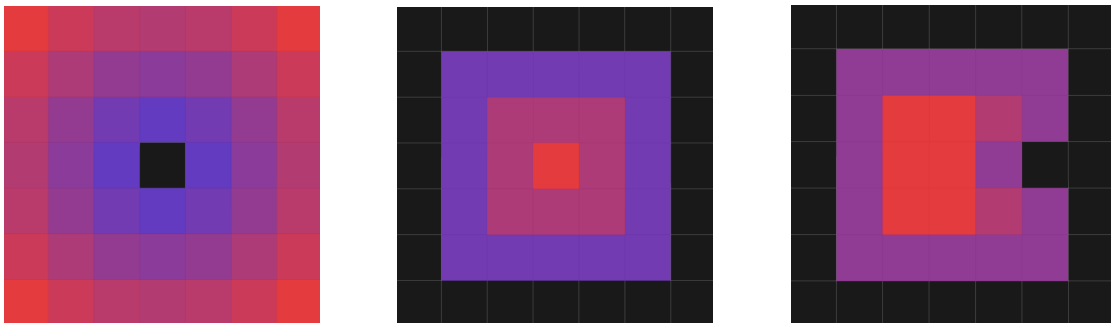
Figure 5.9: An example that shows the need to take obstacles into account when finding a trajectory. In (a) the trajectory does not not take obstacles into account, whereas in (b) it does, by using the lookup table that is visualised in (c).

algorithm called the Euclidean Distance Transform (EDT).

Euclidean distance transform

The mechanism used to prevent trajectories such as the one in Figure 5.9 (a) is a constraint that specifies the minimum distance to the nearest obstacle. Using the EDT, the distance from any point on the map to the nearest obstacle can be calculated. This is done once for the entire occupancy map, and the result is stored in a lookup table.

Figure 5.9 (c) shows the result when applying the EDT to an occupancy map, where blue indicates points in the map that are closer to obstacles, and red indicates points in the map that are further from obstacles. A set of smaller scale examples can be seen in Figure 5.10, where the black tiles are the obstacles. Notice in Figure 5.10 (a) how the diagonal tiles have a different cost than the adjacent tiles, due to the Euclidean distance being used as opposed to the Manhattan distance.



(a) (b) (c)

Figure 5.10: Example lookup tables that are produced when using the EDT, with the obstacles represented by the black tiles.

5.2. Trajectory execution

The trajectory is executed by sampling the optimal velocity commands calculated by the trajectory optimiser and sending them to the vehicle's velocity controller. The trajectory optimiser executes at a frequency of 1 Hz and the velocity sampler executes at a frequency of 10 Hz.

5.3. Summary

This chapter presented the design of the trajectory tracking module, starting with the trajectory optimisation process and ending with the approach used to execute the optimised trajectory. The trajectory optimisation process was designed in an incremental fashion, by first only assuming holonomic constraints, and thereafter adding nonholonomic vehicle constraints, and finally obstacle avoidance constraints. The nonholonomic vehicle constraints specified that the velocity commands which are generated should adhere to the kinematic constraints of the vehicle. This was accomplished by adding the kinematic constraints of the vehicle to the formulation of the trajectory optimisation problem. The obstacle avoidance constraints specified that the trajectory optimiser should find velocity commands that will prevent collisions with the static environment when deviations from the reference trajectory occur. This was achieved by generating a EDT lookup table which can be used to specify the minimum allowed distance between the vehicle and the nearest static obstacle at any point in the optimised trajectory. The trajectory is executed by sending the velocity commands calculated by the trajectory optimiser to the vehicle's velocity controller at a fixed frequency.

Chapter 6

Physical Vehicles

One of the outcomes of this project is that the algorithms that were developed should be tested in a practical setup using physical vehicles. Figure 6.1 show the three vehicles that were constructed for the practical tests. A closer view of one of these vehicles is shown in Figure 6.2, highlighting the major subsystems and components. Figure 6.3 shows the individual parts used for each vehicle before the assembly, and the Bill of Materials can be found in Appendix A.

The first phase of the vehicle design was choosing suitable hardware, including the chassis and wheels of the vehicle as well as the motors used to actuate the vehicle. The second phase in the design consisted of choosing the appropriate electronics to power and drive the motors. This included the batteries, power distribution system, motor driver modules and onboard computer. The final phase of the vehicle design was to develop the software for the onboard computer to receive commands and to actuate the stepper motors to control the translational and rotational velocities. The hardware, electronic, and software designs are described in the following sections.

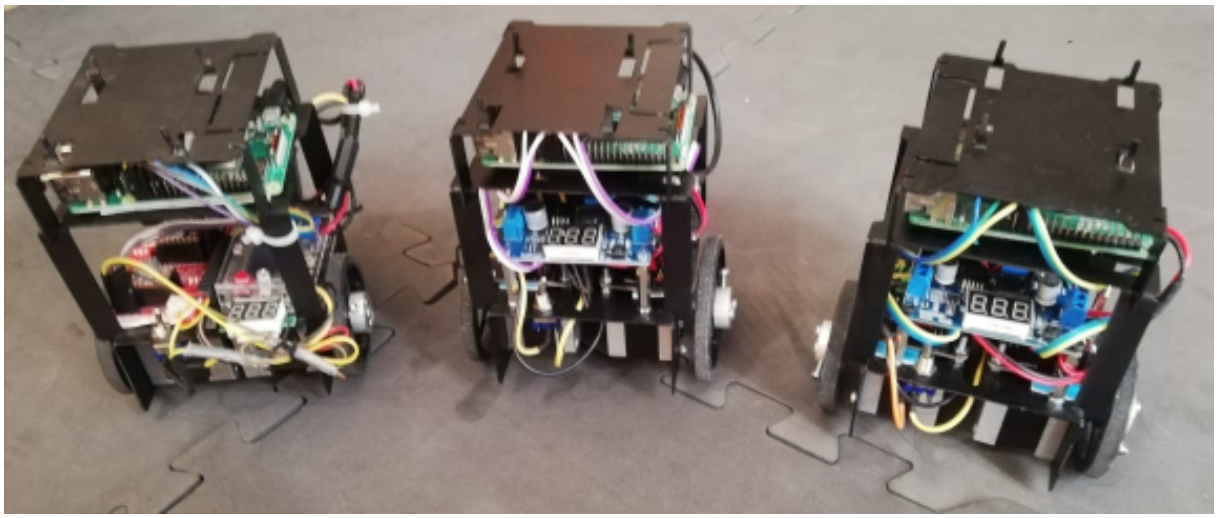


Figure 6.1: The three vehicle built and used to perform the practical tests.

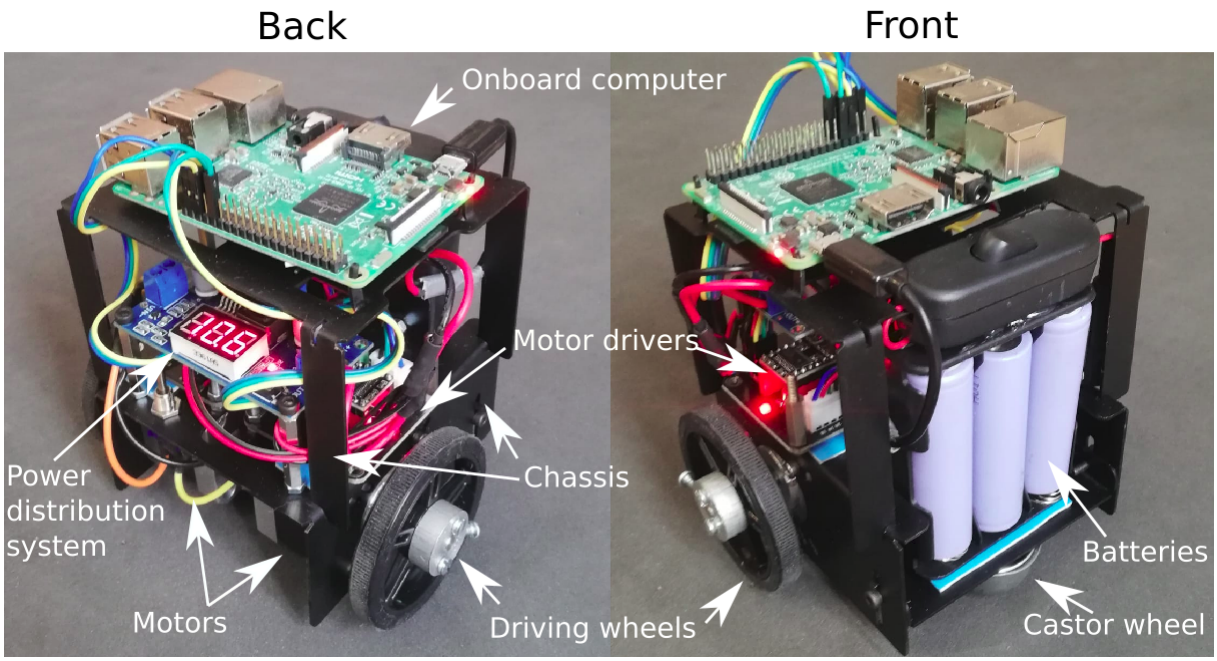


Figure 6.2: Picture of vehicle highlighting major sub-systems.

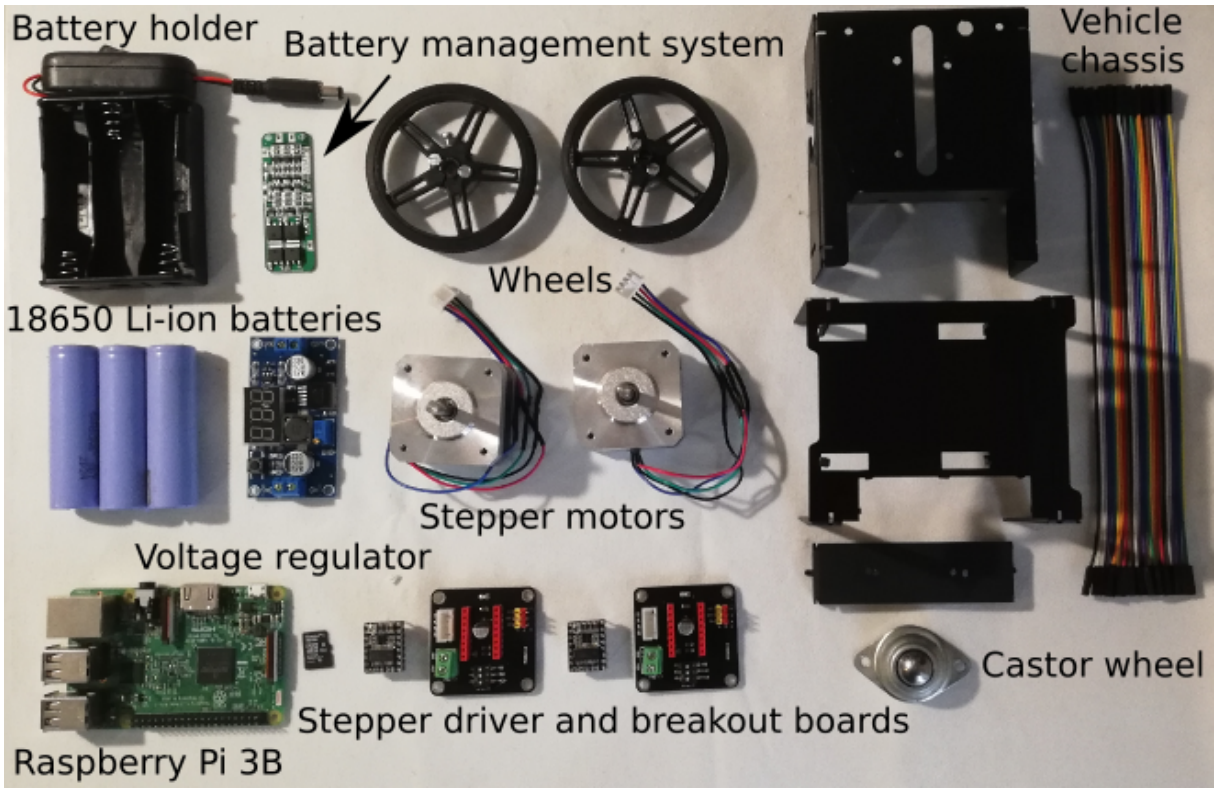


Figure 6.3: The parts used to build the vehicles.

.

6.1. Hardware Design

This section presents the design of the vehicle hardware, which includes the vehicle motors, wheels, and chassis. First, the vehicle requirements are captured and a concept design

is proposed. Then, the component selection is performed from the available options, supported by some design calculations.

The hardware requirements of the vehicle are as follows:

- The vehicle must have two independently controlled wheels, one on either side of the vehicle, according to the differential-drive system.
- Each individual wheel should be able to maintain a maximum speed of $1\,\mathrm{m\,s^{-1}}$.
- The vehicle must have a chassis that can house the necessary electronics. This chassis should be compact, so as to reduce the footprint of the vehicle.

The concept design of the vehicle hardware is presented in Figure 6.4. A cube-shaped chassis is chosen as it is a compact way of housing the vehicle's electronics. The driving wheels are placed in a off-centre configuration, with a castor wheel used for balancing the vehicle.

6.1.1. Choosing the correct motors

The two options considered for the vehicle's motors are brushed DC motors and stepper motors, as shown in Figure 6.5 (a) and (b). The advantage of DC motors is that they are significantly cheaper, and easier to use. The disadvantage of using DC brushed motors is that a feedback control system must be implemented to control their speeds accurately. The angular rate and direction of a brushed DC motor can be controlled in an open-loop fashion by applying an analogue voltage or Pulse Width Modulation (PWM) voltage

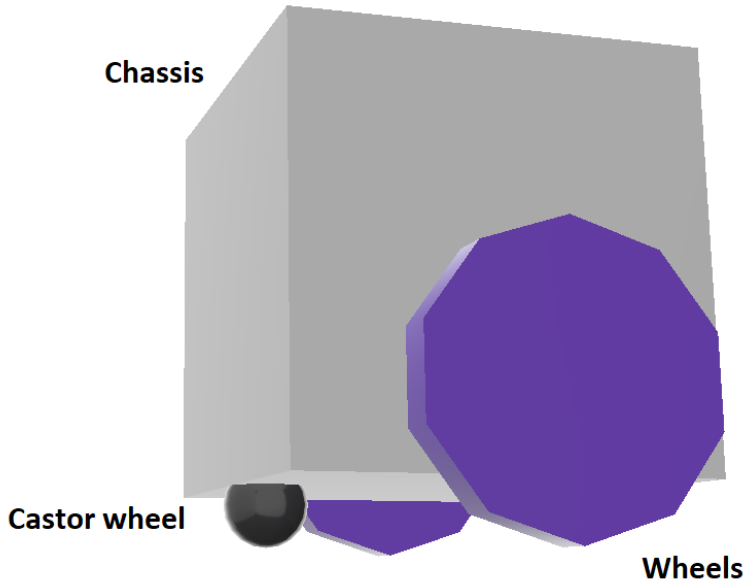


Figure 6.4: The vehicle hardware concept design.

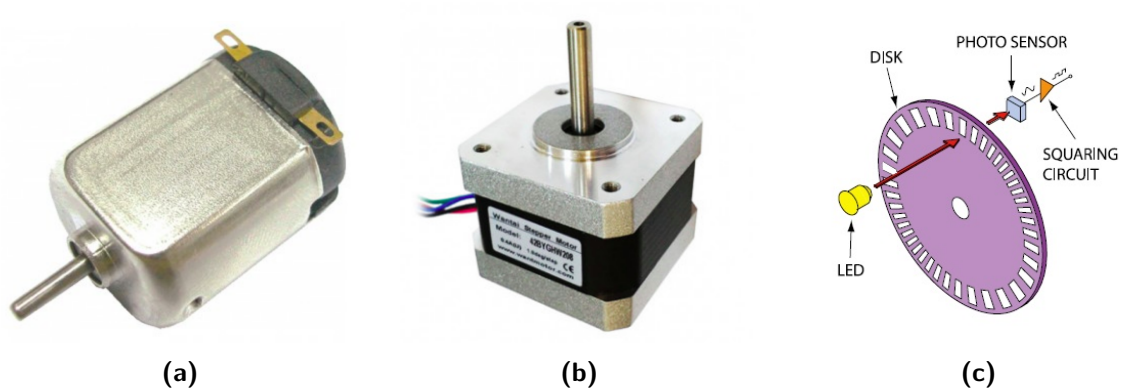


Figure 6.5: A brushed DC motor can be seen in (a), and a stepper motor in (b). The workings of an optical encoder can be seen in (c) (Encoder 2020).

signal to its terminals. However, this open-loop method does not provide any disturbance rejection or robustness to parameter uncertainty.

In order to accurately control the speed of the motor it is necessary to implement a feedback control system, using a sensor that can measure the wheel rotations. One such sensor is the encoder shown in Figure 6.5 (c), which uses optics to determine the speed and direction of the wheel's rotation. Using the encoder adds extra complexity, and is only as accurate as the resolution of the encoder. The advantage of the stepper motor is that accurate open-loop angular rate control can be achieved by commanding the stepping frequency of a square wave that is applied to the terminals of the stepper motor. The stepper motor signal is typically actuated by a dedicated stepper motor driver, which in turn provides an interface through which the direction of rotation and stepping frequency can be commanded. Each step corresponds to a fixed stepping angle, and the number of steps per second therefore translates to a corresponding angular rate. The disadvantage of using a stepper motor is that they are more expensive. Given the relative simplicity of the stepper motor speed control compared to the DC motor, the stepper motor was selected as the most suitable motor for the vehicles.

The NEMA 17 stepper motor and the Pololu a4988 stepper motor driver were selected for the vehicle. The NEMA 17 stepper motor has a step angle of 1.8 degrees (200 steps per revolution). The Pololu a4988 stepper motor driver implements microstepping, which allows the stepper motor to be stepped in increments of 1/16th of the step angle. This increases the stepper motor's stepping resolution by a factor of 16 to 0.1125 degrees (3200 steps per revolution). For low torque applications, the motor's speed can exceed 1000 RPM, which is more than sufficient for this project (PBC Linear 2020).

The stepper motor driver provides DIR and STEP input pins through which the direction and stepping frequency of the stepper motor can be commanded. (A low-to-high transition on the STEP input pin advances the motor one increment.) The DIR and STEP pins will be controlled by the vehicle's onboard computer.

(a) (b)

Figure 6.6: The wheels used to drive the vehicle can be seen in (a), and (b) shows the castor wheel used for balancing the vehicle.

6.1.2. Wheels

Since the type of vehicle was already chosen to be differential-drive, that meant that two wheels had to be used, one on either side of the vehicle. These wheels can be seen in Figure 6.6 (a), also showing the coupling mechanism used to attach the wheels to the motors. The wheel shown in Figure 6.6 (b) is a castor wheel, and is used to balance the vehicle, so that the chassis of the vehicle does not come into contact with the ground.

A wheel radius of 6 cm was used for the vehicles. The angular speed of each wheel needed to achieve a vehicle speed of $1\,\mathrm{m\,s^{-1}}$ can be determined in the following way:

$$\begin{aligned} \omega &= \frac{v}{r} \\ \omega &= \frac{1}{0.06} \\ \omega &= 16.67 \end{aligned} \tag{6.1}$$

This translates to an RPM of 160, which is easily achievable by the chosen stepper motors (PBC Linear 2020).

6.1.3. Chassis

The requirements of the chassis are as follows:

- The chassis should provide the mechanical support for the onboard computer, power distribution system, batteries and stepper motors drivers.
- The chassis should have mounting holes which can be used for the electronics, stepper motors and the castor wheel.

- The chassis should allow for stacking multiple layers of electronics.
- The chassis should be compact so as to reduce the footprint of the vehicle.
- The chassis should be light and easily portable.
- The chassis should be modular, so that it can be easily assembled and disassembled.

The choice had to be made whether a chassis would be bought off-the-shelf, or if it would be manufactured. When no suitable chassis was found which met the requirements, a chassis was designed by a drafter (Mr Cassidy De Wet) according to the specified requirements. The final design of this chassis is shown in Figure 6.7, and consists of four layers, which can be stacked on one another. This design allows for a modular chassis, which can be easily assembled and disassembled through the use of a friction-fit mechanism. Once the design was complete, three chassis were manufactured, one for each vehicle. The chassis were manufactured using 0.9 mm mild steel plating.

6.2. Electronics Design

Once the hardware design phase was completed, the next step was to design the electronics system. The following requirements were identified for the electronics system:

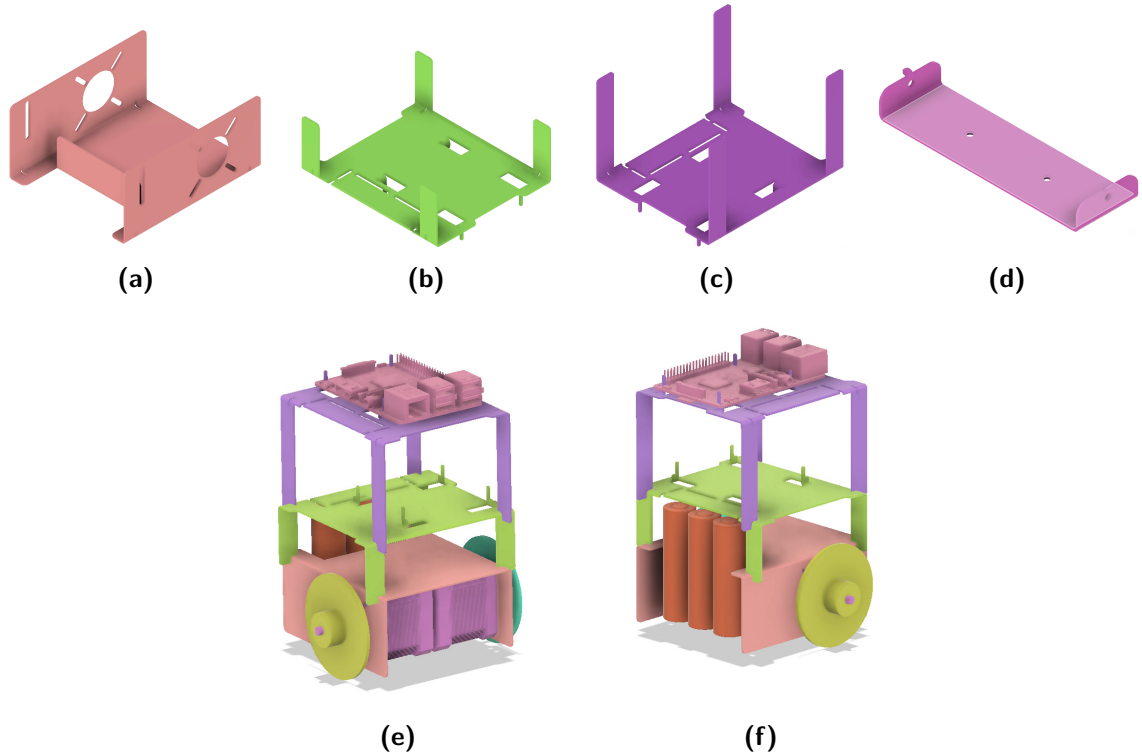


Figure 6.7: The individual components of the vehicle chassis can be seen in (a) through (d), with the assembled presentation in (e) and (f).

- There should be an onboard computer, which is capable of receiving and executing rotational and translational velocity commands.
- The vehicle must have batteries which are able to power all the electronics, including the onboard computer and the stepper motors.
- The batteries must enable the vehicles to be operated for at least one hour between charges.
- The batteries must be rechargeable, so that they can be reused for several practical tests.
- There must be voltage regulators which are able to convert the battery voltages to the voltages required by the electronic components.
- There should be stepper drivers which are able to provide an interface between the onboard computer and the stepper motors.

The architecture of the vehicle electronics is shown in Figure 6.8. The major components of the electronics are the onboard computer, the power distribution system and the stepper motor drivers. The design of these components are presented in the following sections.

6.2.1. Onboard computer

The purpose of the onboard computer is to listen for rotational and linear velocity commands, and then to execute these commands by controlling the stepper motors. The stepper motors are controlled through the stepper drivers, which are interfaced by sending

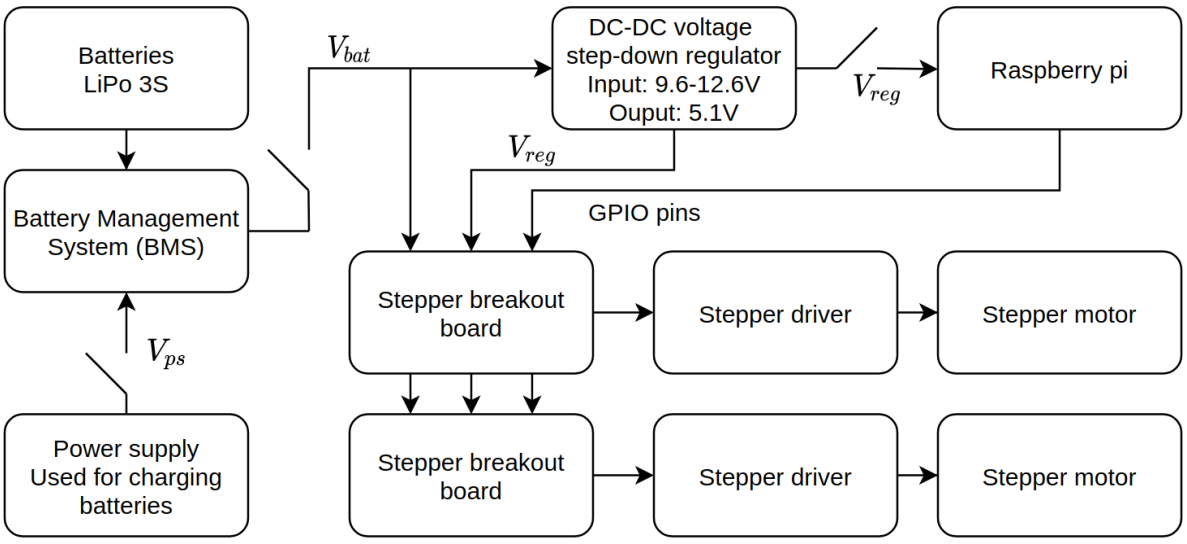


Figure 6.8: The complete electronics architecture, showing the power distribution system and the General-Purpose Input/Output (GPIO) connection between the Raspberry Pi and the stepper motor driver boards.

digital HIGH and LOW signals on the DIR, STEP and ENABLE pins of the stepper drivers. The onboard computer must have the Ubuntu 16.04 operating system installed, so that it can receive the velocity commands using ROS Kinetic. The Raspberry Pi 3B was chosen as the onboard computer for the vehicle, as it has all the necessary GPIO pins, is well documented, has an extensive user base, and can run ROS Kinetic using the Ubuntu 16.04 operating system.

6.2.2. Power distribution system

The following requirements were identified for the power distribution system:

- The power must be supplied by batteries, which allow for at least one hour of vehicle operating time between charges.
- The batteries must supply enough current for all the electronics.
- The battery voltage must be converted into the required supply voltage of the onboard computer.
- The batteries must be rechargeable.

In order to choose suitable batteries, it was necessary to consider the current draw of the electronic components. When doing this, only the current draw of the onboard computer and stepper motors were taken into account, as they were the only components which significantly contributed to the overall current draw of the vehicle. The chosen onboard computer was the Raspberry Pi 3B, which has a rated current draw of less that 500 mA (RasPi.TV 2016). The stepper motors that were used has a rated current draw of 800 mA each (Micro Robotics 2020). The batteries should therefore be able to provide a total of 2100 mA over a duration of one hour. This requires the capacity of the batteries to be at least 2100 mA h.

The other aspect to the battery selection which has to be considered is the supply voltage of the electronic components. Two different voltage levels are used in the vehicle electronics, one for the Raspberry Pi and one for the stepper motors. The Raspberry Pi requires an supply voltage of 5.1V, whereas the stepper motors can operate within a range of supply voltages. The a4988 stepper motor driver specifies a motor voltage range of 8V to 35V. Using this information, it was decided that the battery used to power the vehicle would be a 3S lithium ion battery configuration. Each lithium ion battery is capable of providing a voltage of 4.2V when fully charged and 3.2V when discharged. When using the 3S configuration, this translates to 12.6V when fully charged and 9.6V when discharged, as three of the batteries are connected in series. This would provide sufficient voltage to power the motors, although a step-down voltage converter has to be used for providing a stable 5.1V for the Raspberry Pi. The individual lithium ion batteries were chosen to have

a capacity of 3000 mA h, so that the total current draw of the vehicle could be maintained for at least one hour. The chosen lithium ion batteries were rechargeable, and an onboard Battery Management System (BMS) was used to ensure the safe discharging and charging of the batteries.

In order to supply the Raspberry Pi with a voltage of 5.1V, it was necessary to use a voltage regulator, as the battery voltage varied between 9.6V when discharged and 12.6V when fully charged. This voltage regulator had to be able to supply the rated 500 mA current draw of the Raspberry Pi 3B. The voltage regulator chosen for this project was the LM2596. This voltage regulator was chosen as it was locally available, and it is able to supply a current of 3000 mA, which is more than sufficient for the Raspberry Pi 3B (Texas Instruments 2020).

6.2.3. Stepper motor drivers

The stepper motor drivers are mounted on stepper driver breakout boards. The stepper motor drivers are commanded by the Raspberry Pi onboard computer through GPIO pins that are connected to the enable, direction, and step inputs of the stepper drivers. The Raspberry Pi outputs logic high or logic low values to the ENABLE and DIR pins to enable or disable the stepper motors, and to select the stepping direction, and outputs a square wave to the STEP input pin to control the stepping frequency. (Each low-to-high transition on the STEP input pin advances the stepper motor one increment.)

The STEP input is connected to one of the Raspberry Pi's PWM-capable pins, and the stepping signals are generated by the Raspberry Pi's PWM hardware peripherals. Although the Raspberry Pi is capable of using software PWM on any of its GPIO pins, only two of its GPIO pins can be used for hardware PWM. The hardware PWM is preferred above the software PWM, since it allows more consistent timing and more accurate control of the stepping frequency.

6.3. Software Design

The onboard computer software implements the velocity controller for the vehicle. The software must listen for incoming translational and rotational velocity commands, must convert them to directions and motor step frequencies for the two stepper motors, and must output the digital and PWM signals that are connected to the stepper motor drivers' enable, direction and step input signals.

The software is implemented as a ROS node in the ROS Kinetic environment, which in turn runs on the Ubuntu 16.04 operating system installed on the Raspberry Pi onboard computer. The ROS node connects to the ROS network via WiFi to listen for velocity commands published to the "/<vehicle_namespace>/cmd_vel" ROS topic. The velocity

commands are typically published by the ROS node that implements the model predictive controller that performs the trajectory execution, or by another ROS node which is used to control the vehicle velocities manually. The ROS Kinetic environment was selected because it was the most supported version of ROS at that time. The Ubuntu 16.04 operating system was used because ROS Kinetic does not support Ubuntu 18.04.

The flow diagram for the velocity controller software is shown in Figure 6.9. The software listens for new velocity commands on the "/<vehicle_namespace>/cmd_vel" ROS topic for the specific vehicle. When a new velocity command is received, the command values are checked. If both the linear and the angular velocity commands are zero, then the two stepper motors are disabled using the ENABLE pins of the two stepper motor drivers. Otherwise, the linear and angular velocity commands are converted to motor step frequencies for the left and right stepper motors. The linear velocity command is converted to the common step frequency for both wheels, and the angular velocity command is converted to differential step frequencies for the two wheels. The motor directions and step frequencies are then commanded by outputting digital signals and PWM signals to the DIR and STEP input pins of the two stepper motor drivers.

The velocity controller ROS node was written in the Python programming language

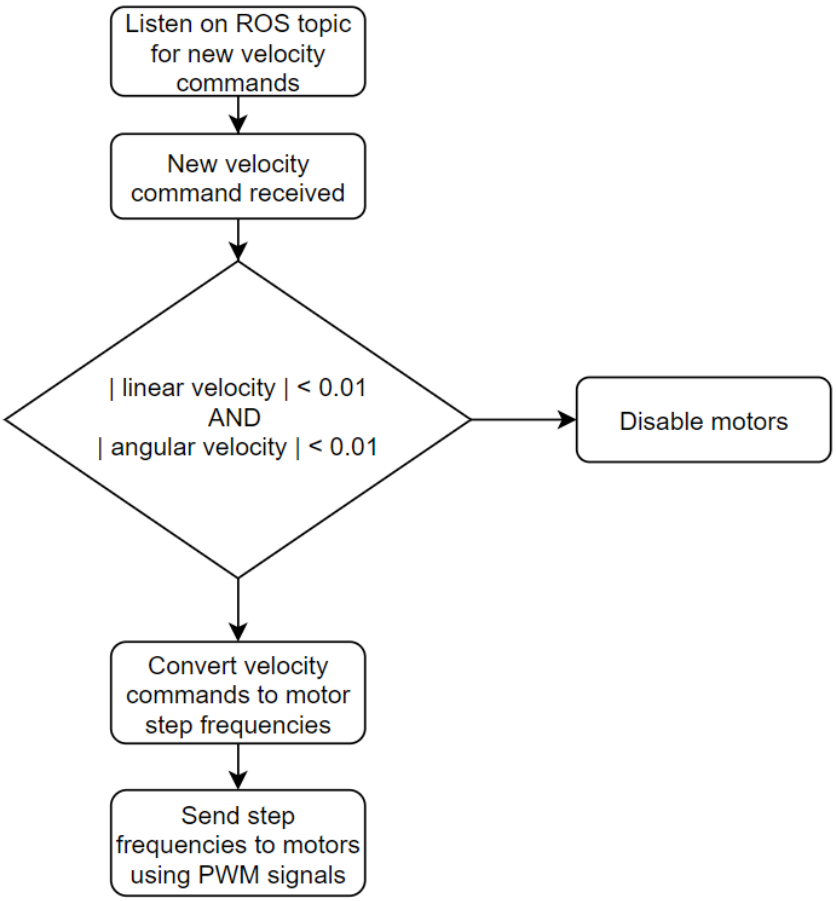


Figure 6.9: The flow diagram of the software used to implement the velocity controller on the Raspberry Pi.

and used the pigpio software library to interface with the hardware PWM pins of the Raspberry Pi.

6.4. Summary

This chapter presented the design and implementation of the physical vehicles that were used to practically test the cooperative navigation system. The high-level concept design was described, followed by the detailed designs of the hardware, electronics, and software. The hardware design covered the stepper motors, the wheels, and the vehicle chassis. The electronic design covered the onboard computer, the power distribution system, the batteries, the voltage regulators and the stepper motor drivers. The software design covered the onboard computer software that implements the velocity controller for the vehicle that listens for linear and angular rate commands, converts them to stepper motor directions and step signal frequencies, and outputs the signals to the stepper motor drivers.

Chapter 7

Vehicle Pose Estimation System

One of the objectives of the project was to test the cooperative navigation algorithms, both in simulation and with a practical setup. To cooperatively plan and execute the vehicle trajectories, the system requires continuous updates of the vehicle poses. When testing the system in simulation, the vehicle pose information is provided by the simulation environment and is readily available. However, when testing the system with the practical setup, the poses of the vehicles must be determined from sensor measurements.

The requirements of the pose estimation system are as follows:

- The pose estimate of the vehicle must be updated at a frequency of at least 10 Hz.
- The accuracy of the pose estimates should be on average accurate to within 3 cm and 5°. These values were chosen based on the fact that trajectory planning allows for the vehicle to deviate from the planned trajectory by up to 8 cm. A 3 cm inaccuracy in the pose estimation leaves a further 5 cm safety margin for actual deviation that might occur during the execution of the trajectory.

This chapter presents the design of the vehicle pose estimation system used for this project. The vehicle pose estimation was accomplished by placing fiducial markers on the vehicles, and then using external cameras and a computer vision algorithm to detect the pose of the markers.

7.1. Pose estimation approach

As the focus of this project is cooperative navigation and not pose estimation, the vehicle pose estimation must be simple yet reliable. The choice was therefore made to perform the pose estimation using an external system instead of each vehicle having its own onboard system. Two different approaches were considered for the external pose estimation system: using the HTC Vive, and using computer vision to detect fiducial markers.

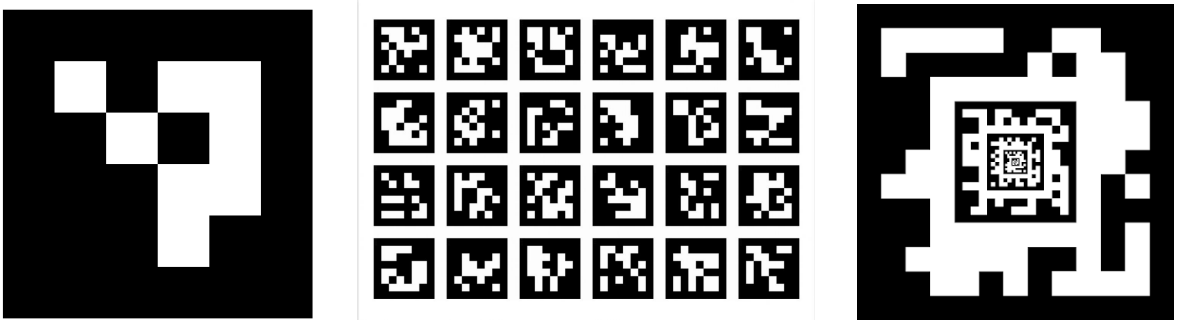
The first approach considered is using the HTC Vive. The HTC Vive is a virtual reality headset that uses an external motion capture system to track the movement of the headset, as well as an assortment of accessories. The HTC Vive system sells separate tracking devices that can be attached to objects and used to perform accurate motion

tracking. Borges *et al.* (2018) described how the HTC Vive can be used with ROS to provide highly accurate tracking at an affordable price, and concluded that the system is able to provide position and orientation measurements accurate to within 13.5 mm and 0.0193° respectively.

The second option considered is using fiducial markers and computer vision detection algorithms to determine the pose of the vehicles. The use of fiducial markers has gained popularity in the robotics community as an inexpensive and easy way of performing object tracking, especially as computer vision algorithms have become increasingly accessible. Although there are several different fiducial tags that could be used to perform object tracking, the ones considered for this project are called ArUco tags (Romero-Ramirez *et al.* 2018, Garrido-Jurado *et al.* 2016), an example of which can be seen in Figure 7.1.

Figure 7.1 (a) shows the most basic ArUco marker consisting of a matrix of black and white tiles. When using the appropriate computer vision software it is possible to infer from a captured image the spatial transform between the camera that captured the image and the marker in the image. One of the disadvantages to using a single marker is that when the marker is viewed from certain angles it is possible to get a false positive result, meaning that the detection module returns an incorrect marker pose. The marker in Figure 7.1(b) presents a more robust solution, and is called an ArUco gridboard. Using a gridboard allows the computer vision software to compare the transforms to all the markers in the gridboard. Providing that the spatial transforms between the markers in the gridboard are specified beforehand, it is able to use this information to detect and discard individual false positive detections.

The third type of marker is shown in Figure 7.1 (c), and is called a fractal marker. This marker is used when it needs to be detected from a range of different distances. This is a challenge when using a normal marker because the detection struggles when the marker is too far away from the camera, and also fails if it is too close as the marker becomes partially occluded. One possible application of the fractal marker is for Unmanned Aerial



(a) (b) (c)

Figure 7.1: Different ArUco marker examples. In (a), the standard ArUco marker can be seen when using the DICT_4X4_50 dictionary. An example of a ArUco gridboard can be seen in (b), as well as a fractal marker in (c).

Vehicle (UAV) autonomous landing procedures. The UAV must be able to detect the marker both when it is very far away from and when it is very close to the platform.

The ArUco fiducial tag approach was chosen as the object tracking solution due to the fact that it is significantly easier and less expensive to implement, and can be easily scaled to track more vehicles by simply printing more tags. The following sections describe the design of the vehicle pose estimation system using this approach.

7.2. Reliable detection of ArUco markers

Each ArUco marker that is used belongs to a predefined dictionary, which is categorised based on the number of tiles present in the ArUco marker. The dictionary used in this project is the DICT_4X4_50 dictionary, which means that the markers have 4X4 tiles, for a total of 16 tiles arranged in a square. The 50 refers to the fact that the dictionary has 50 predefined marker types. Each different marker type has a unique pattern of black and white tiles.

The detection module needs to know the dictionary that is used, as well as the side length of the markers. The dictionary is used to match each marker to its ID, and to map the marker ids to the objects to which they belong. For example, if marker ID 0 is used to represent vehicle one, then the transform between the camera and marker ID 0 would represent the transform between the camera and vehicle one. The physical length of the marker is used as a scaling factor to accurately determine the transforms. If the marker length is incorrectly specified, the transform will be inaccurate by the same scaling factor as the incorrect marker length to the actual marker length.

One of the challenges with using ArUco markers for detection is that they suffer from the ambiguity problem. This problem is discussed in the detailed ArUco documentation found on their website (ArUco 2020), and is illustrated in Figure 7.2. This problem results in the possibility of false positive measurements, where the detection module is unsure as to the orientation of the marker. There are many ways of addressing this problem, but the approach used in this project is to make use of the ArUco gridboard functionality. Using the ArUco gridboards, it is possible to specify a collection of markers as belonging to one gridboard. Contrary to the name it is possible for the markers belonging to the gridboard to be arranged in any way, providing the spatial transforms from each of their corners to a relative point is specified. For this project it was chosen to arrange the ArUco markers in a five-sided box that could be placed over the vehicles, allowing for reliable detection and unambiguous pose estimation from all possible different angles. The ArUco boxes that were mounted on the vehicles are shown in Figure 7.3.

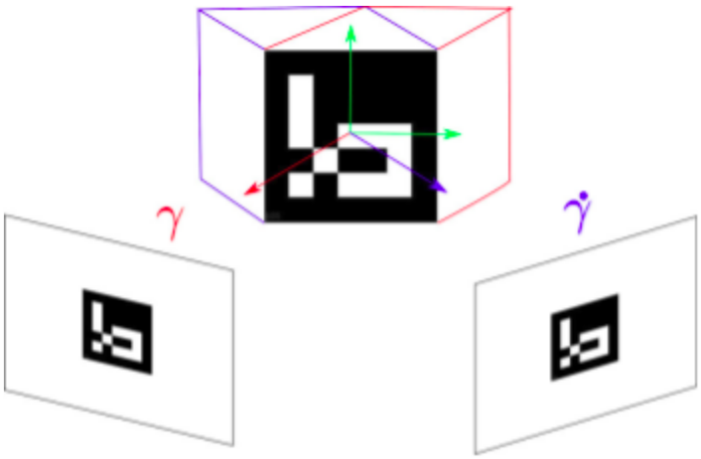
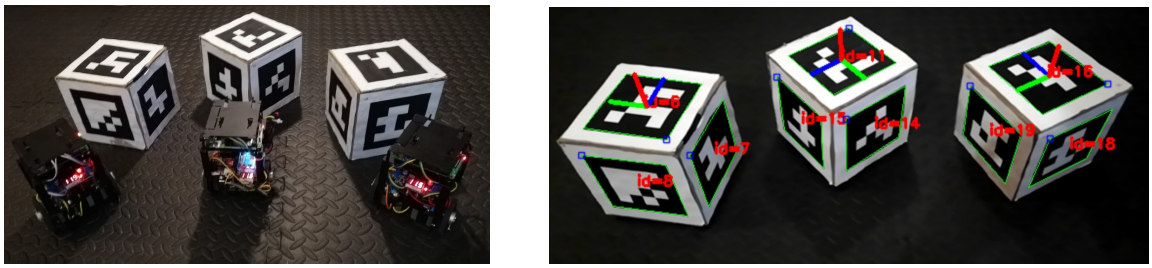


Figure 7.2: An example that shows the ambiguity problem of ArUco markers. Notice how the same marker detection can be interpreted as two different transforms (ArUco 2020).



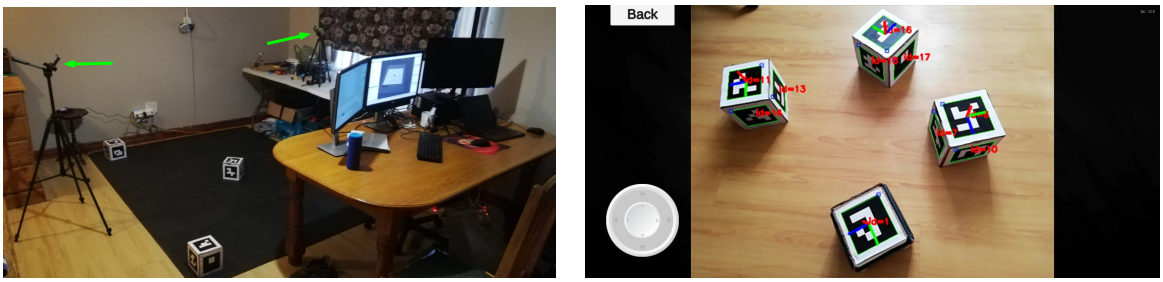
(a) **(b)**

Figure 7.3: The pose of the vehicles are tracked by placing an ArUco box over them, and then using the ArUco gridboard detection algorithm to detect the pose of the box.

7.3. Practical setup

Once the ArUco markers could be reliably detected, the next step was to create a setup which could be used to perform the practical test. The final version of this setup is shown in Figure 7.4 (a), with the camera positions indicated by the green arrows. Two cameras were used in this setup, as it improved the total coverage as well as the frequency at which pose estimates could be obtained.

Two Android smartphones were used as the cameras, the one being a Huawei P20 Lite and the other a Sony Xperia Z3 Compact. A software application was developed which allowed the smartphones to perform the ArUco detection and publish the resulting pose estimate to the appropriate ROS topic. The application was developed using the Unity3D framework, and the interface is shown in Figure 7.4 (b). Aside from performing the ArUco detection, the application could also be used to send manual velocity commands to the vehicles, and to perform a camera calibration procedure. Both of the smartphones used were able to perform the ArUco marker pose detection at a frequency of more than 15 Hz,



(a) (b)

Figure 7.4: The practical setup that was used for testing the algorithms can be seen in (a), with the green arrows indicating the position of the cameras. The cameras used were Android smartphones, running an application developed using the Unity3D framework. In (b), the application interface used for the smartphone cameras can be seen.

well within the required vehicle pose update frequency of 10 Hz.

7.4. Spatial realignment

Both the cooperative trajectory planning and the trajectory tracking algorithms require the pose of the vehicles to be expressed relative to a fixed reference frame that represents the world axis system. This presented a problem, as the ArUco marker pose estimation algorithm returns the transforms relative to the pose of the camera. This problem is exacerbated when considering that more than one camera is used, so the estimated pose of each vehicle would vary based on which camera performs the pose estimate.

In order to mitigate this problem, the transforms returned by the ArUco detection algorithms are processed so as to form a transform tree relative to a fixed point, using a process referred to as spatial realignment. During this process, the transforms are realigned to find the transforms from the fixed reference frame to each of the vehicles as well as to the cameras. This allows the pose estimates from the individual cameras to be fused into a single pose estimate. This process is shown for both camera one and two in Figure 7.5. The fixed reference frame is defined by using another ArUco box, as indicated by the green arrows in Figure 7.5 (a) and (d). Once the correct transform tree has been determined for both cameras, these are then fused into a single transform tree, containing the fused pose estimates from both cameras. This fused transform tree can be seen in Figure 7.6, showing the pose of all three vehicles as well as the two cameras relative to the reference point.

In order to perform the transform tree realignment process, it is first necessary to express each pose as a transform matrix of the follow form:

$$T = \begin{pmatrix} r_{1,1} & r_{1,2} & r_{1,3} & t_x \\ r_{2,1} & r_{2,2} & r_{2,3} & t_y \\ r_{3,1} & r_{3,2} & r_{3,3} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.1}$$

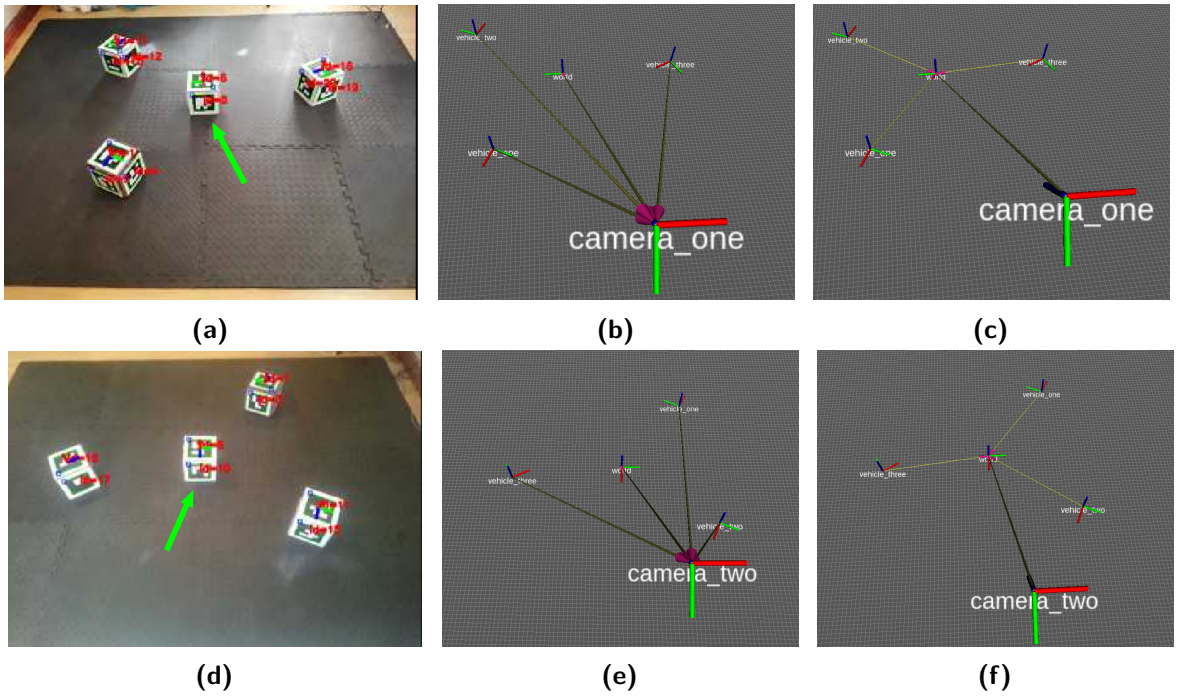


Figure 7.5: These figures show the process whereby the transforms returned by the ArUco detection module are changed so that they are relative to the reference marker, indicated by the green arrows in (a) and (d). The realignment process for camera one can be seen in (a) through (c), whereas that of camera two is shown in (d) through (f).

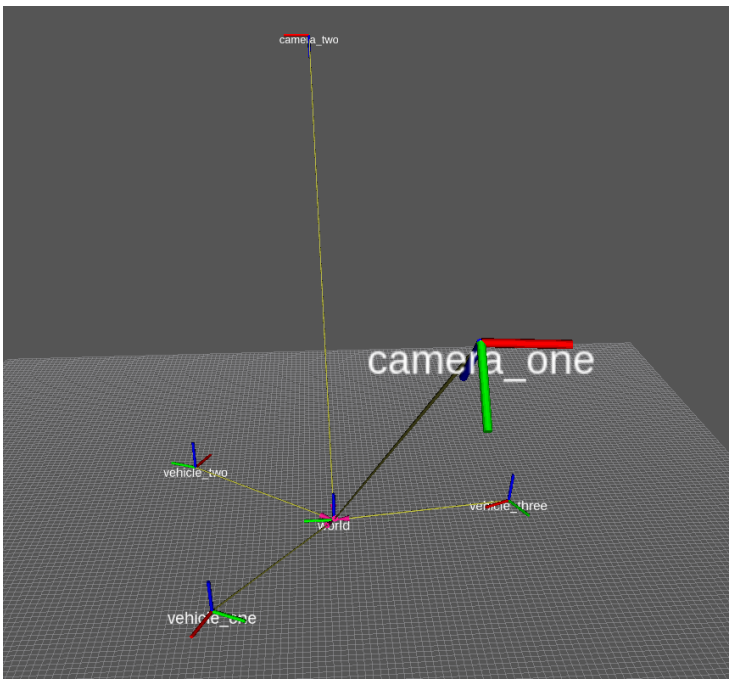


Figure 7.6: The complete transform tree made by fusing the measurements from both cameras after they have been correctly processed.

where $r_{i,j}$ are the elements of the rotation matrix and t_x,t_y and t_z are the elements of the translation vector. Since the rotation supplied by the detection module is in the quaternion format, it must first be converted to a rotation matrix. This can be done using the following equation:

$$\begin{aligned} R &= \begin{pmatrix} r_{1,1} & r_{1,2} & r_{1,3} \\ r_{2,1} & r_{2,2} & r_{2,3} \\ r_{3,1} & r_{3,2} & r_{3,3} \end{pmatrix} \\ &= \begin{pmatrix} 1-2y^2-2z^2 & 2xy+2wz & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz+2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{pmatrix} \end{aligned} \tag{7.2}$$

where w, x, y and z are the four elements of the quaternion.

Once the transforms have been expressed in terms of the transformation matrices, the transform tree can be rearranged, as shown in Figure 7.7, where the black arrows represent the transforms published by the ArUco detection algorithm, and the blue arrows are the desired transforms. Now that the transforms are expressed as matrices, finding the desired transforms can be done easily, as shown in the following example, where the transform between the origin and vehicle_one is found.

$$T_{OV1} = T_{C1O}^{-1} T_{C1V1} \tag{7.3}$$

where T_{OV1} is the pose of vehicle_one relative to the world reference frame, T_{C1O} is the pose of the world reference frame relative to camera_one, and T_{C1V1} is the pose of vehicle_one relative to camera_one.

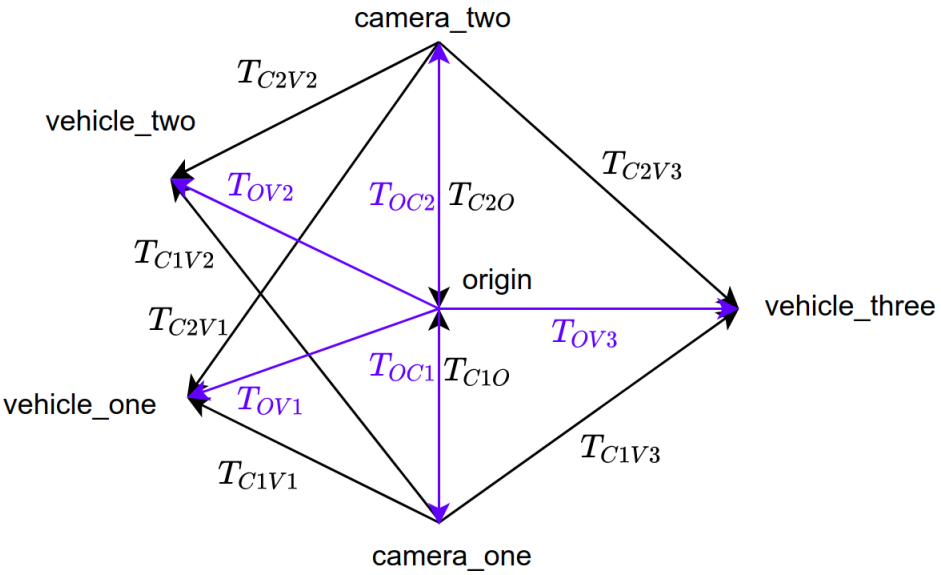


Figure 7.7: The transform trees from camera one and two shown in black, as well as the desired transform tree shown in blue.

7.5. ArUco pose estimation accuracy

Once the spatial realignment process had been incorporated, it was possible to evaluate the accuracy of the ArUco marker detection and pose estimation process. This was done by performing sixteen ArUco pose estimates and comparing the ground truth values with the estimated values. The sixteen poses chosen for this test is shown in Table 7.1 along with the estimated poses from the ArUco detection system. Figure 7.8 (a) shows these results visually, with the dark blue arrows indicating the ground truth poses and the lighter blue arrows indicating the measured poses. Figure 7.8 (b) presents the statistical results of these tests, showing an average position and orientation measurement accuracy of within 2 cm and 2°, which meets the required accuracy of 3 cm and 5°.

7.6. State estimator

Once the vehicle pose measurements have been transformed to the common reference frame, the next step is to use the pose measurements together with the vehicle's velocity commands to iteratively estimate the vehicle's pose. To this end, the state estimator shown in Figure 7.9 is used. There are two main steps to the estimation process, namely prediction and correction. In the prediction step, the pose of the vehicle is propagated forward in time from the previous pose estimate using the vehicle's dynamic model and the velocity commands that were received at the previous sampling instant. The pose is

Table 7.1: Ground truth poses and measured poses for ArUco accuracy test.

Actual pose			Estimated pose		
$x[m]$	$y[m]$	θ [°]	$x[m]$	$y[m]$	θ [°]
0.500	0.500	135.0	0.497	0.525	139.8
0.500	0.500	45.0	0.489	0.503	50.2
0.500	0.500	-45.0	0.495	0.518	-44.1
0.500	0.500	-135.0	0.482	0.523	-128.0
-0.500	0.500	135.0	0.506	0.501	135.0
-0.500	0.500	45.0	-0.505	0.505	46.9
-0.500	0.500	-45.0	-0.524	0.509	-39.8
-0.500	0.500	-135.0	-0.514	0.489	-132.0
0.500	-0.500	135.0	0.507	-0.494	133.4
0.500	-0.500	45.0	0.500	-0.479	45.4
0.500	-0.500	-45.0	0.494	-0.481	-44.8
0.500	-0.500	-135.0	0.486	-0.498	-138.0
-0.500	-0.500	135.0	-0.487	0.515	131.9
-0.500	-0.500	45.0	-0.473	0.506	42.9
-0.500	-0.500	-45.0	-0.489	-0.519	-46.2
-0.500	-0.500	-135.0	-0.484	-0.501	-136.9

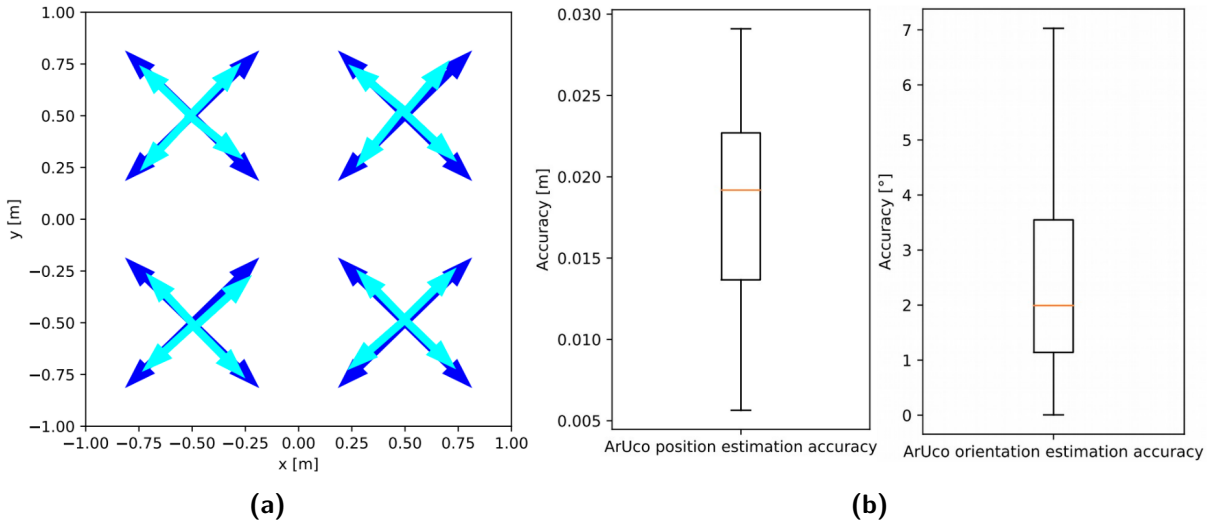


Figure 7.8: The results from performing a series of ArUco detections at four different positions, with four orientations at each position, giving a total of sixteen poses. In (a), the ground truth poses are indicated by the dark blue arrows, whereas the estimated poses are shown with the light blue arrows. The statistical analysis of the marker detections can be seen in (b), showing a position and orientation measurement average accuracy of within 2 cm and 2° respectively.

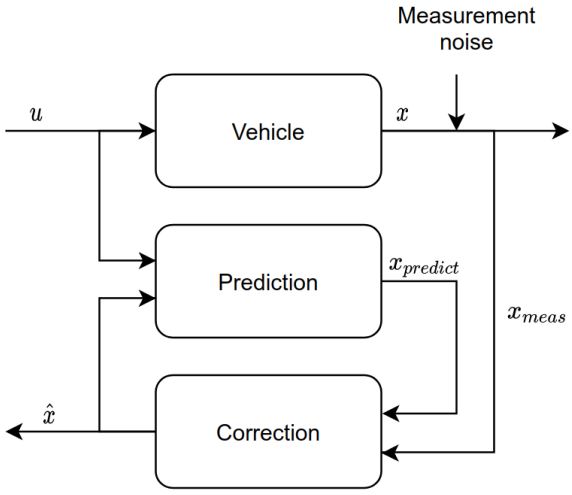


Figure 7.9: The block diagram of the state estimator used for finding the vehicle poses.

predicted using the following equation:

$$\begin{pmatrix} x_{t+\Delta t} \\ y_{t+\Delta t} \\ \theta_{t+\Delta t} \end{pmatrix} = \begin{pmatrix} \Delta t \cos\theta_t & 0 \\ \Delta t \sin\theta_t & 0 \\ 0 & \Delta t \end{pmatrix} \begin{pmatrix} v_t \\ w_t \end{pmatrix} + \begin{pmatrix} x_t \\ y_t \\ \theta_t \end{pmatrix} \tag{7.4}$$

where v_t an w_t are the linear and angular velocity commands received at the previous sampling instant, and Δt is the sampling period. The second step of the estimation process is the correction, which is performed every time a measurement update is received. This stage uses a weighted sum of the predicted and measured poses of the vehicle to update

the pose estimate, as shown below:

$$\hat{x} = \alpha x_{predict} + \beta x_{meas} \tag{7.5}$$

where α and β are the prediction and correction weightings respectively and $\hat{x}$ is the estimated pose of the vehicle. Figure 7.10 shows the result of using the state estimator, with the measured state of the vehicle shown in (a), and the estimated values shown in (b), (c) and (d) for different weightings. These measurements were collected by manually controlling one of the vehicles and driving around the testing area, recording the measurements published by the cameras. From the raw measured data in Figure 7.10 (a) it is possible to distinguish between the measurement updates from the two different

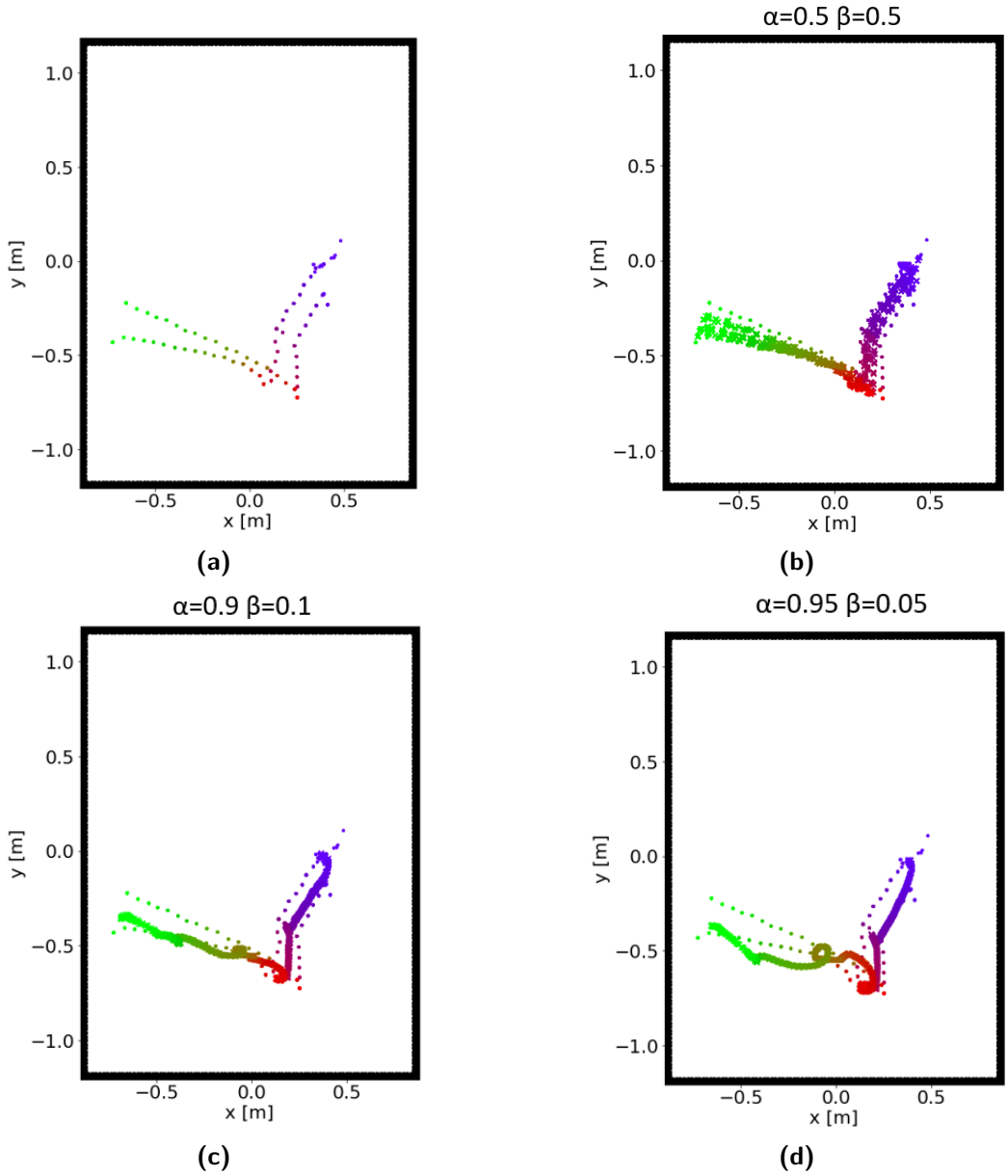


Figure 7.10: The raw measured poses of the vehicles is seen in (a), with the estimated states of the vehicles shown in (b), (c) and (d) for different correction weightings used.

cameras, as there are two clear measurement groupings which are indicated by the dashed lines. Figure 7.10 (b) shows the result of using a weighting of 0.5 for both α and β, with significant scattering still present in the estimated pose. Figure 7.10 (b) and (c) shows the result of using a larger weighting for α, favouring the predicted pose over the measured pose of the vehicle. Based on these results, it was decided to use the weightings of 0.9 and 0.1 for α and β respectively, as in Figure 7.10 (c).

7.6.1. Addressing Euler wrapping

When performing the correction step a strange behaviour emerged, causing the estimated pose of the vehicle to include unnecessary rotations. During the driving segment shown in Figure 7.11, the vehicle moved in a straight line without rotating, which means that recorded state estimation was erroneous. At first this seemed to be caused by measurement updates with incorrect orientations, but this was not found to be the case, as the orientations of the measurements were all pointing in the expected directions. After some investigation, it was found that the erroneous state estimation shown in Figure 7.11 resulted from the Euler angles wrapping around at +/- 180°.

As an illustration, consider the following example, where the predicted orientation of the vehicle is 175°, and the measured orientation of the vehicle is −175°. Intuitively, the estimated orientation of the vehicle should by roughly 180°, but this is not the case. When using the α and β weightings of 0.9 and 0.1, this results in an estimated angle of 140° according to the following equations:

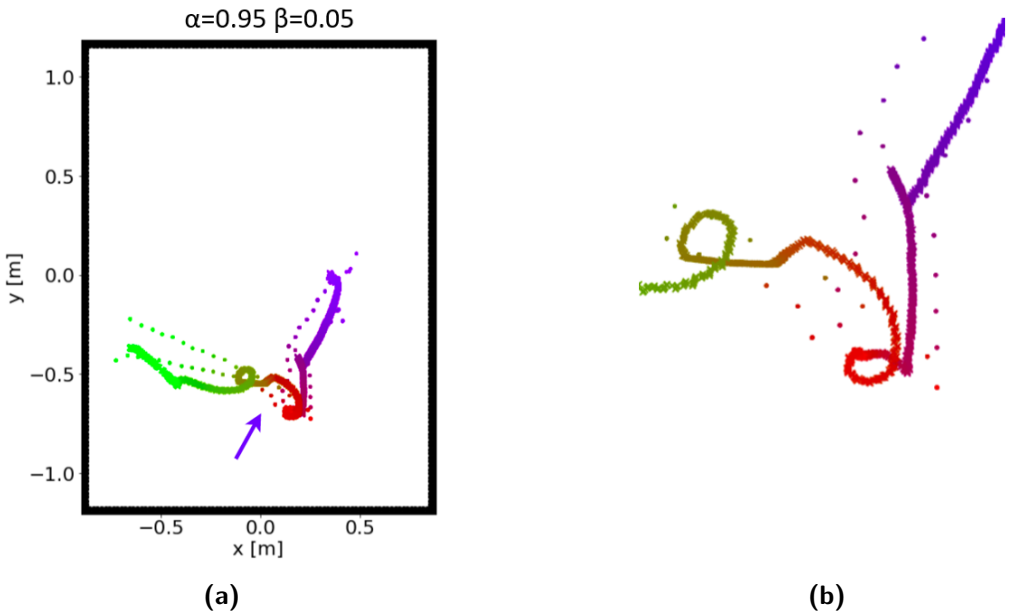


Figure 7.11: An example showing the erroneous estimated state of the vehicle caused by using Euler angles. The blue arrow in (a) shows an example of the erroneous estimated state, and (b) shows a closer view of the error region.

$$\begin{aligned}\hat{x} &= \alpha x_{predict} + \beta x_{meas} \\ &= (0.9)(175°) + (0.1)(-175°) \\ &= 140°\end{aligned} \tag{7.6}$$

This is clearly the wrong answer, and can be avoided by rather representing the orientation of the vehicle as unit vectors using complex numbers when performing the correction step, as follows:

$$\begin{aligned}\hat{x} &= \alpha x_{predict} + \beta x_{meas} \\ &= (0.9)(1\angle 175°) + (0.1)(1\angle -175°) \\ &= (0.9)(-0.996 + 0.087i) + (0.1)(-0.996 - 0.087i) \\ &= -0.996 + 0.0696i \\ &= 1\angle 176°\end{aligned} \tag{7.7}$$

This gives an orientation of 176°, which is what would be expected. Once this change is made, the filter correctly outputs the estimated pose of the vehicle, as seen in Figure 7.12.

7.7. Summary

This chapter presented the design and implementation of an external vision-based system to track the poses of the vehicles during the practical tests. The system accomplishes this by using two cameras and a computer vision algorithm which can detect the poses of five-sided ArUco fiducial marker boxes placed over the vehicles. After evaluating the accuracy of the ArUco detection it was found that the system is able to provide position

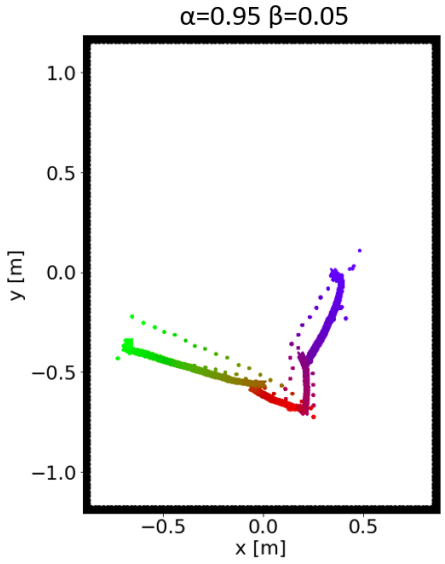


Figure 7.12: The fixed state estimator after complex numbers were used for the correction step.

and orientation measurement updates for each vehicle to within 2 cm and 2° respectively, meeting the required specification of 3 cm and 5°. The pose estimation system is also able to update the estimated pose of the vehicles at a frequency of more than 15 Hz, which met the required pose update frequency of more than 10 Hz. These measurements are used by a state estimation filter, which finds and publishes the estimated pose of the vehicles.

Chapter 8

System Integration and Results

This chapter describes the process used to verify the performance of the cooperative navigation algorithms. The chapter starts by giving an overview of the tests that were performed as well as a description of the the simulation and practical environments used when evaluating the algorithms, followed by a brief discussion on the results that were obtained. The chapter then presents a more detailed discussion on the results from the individual tests that were performed. The first tests were aimed at evaluating the performance of the cooperative trajectory planning module, after which the accuracy of the real vehicles' velocity controllers are tested. Finally, the complete system performance of the cooperative navigation algorithms are evaluated both in simulation and using a practical setup.

8.1. Overview of test environments

The goal of this project was to develop a system that would allow the cooperative navigation of multiple Autonomous Ground Vehicles (AGVs). To this end, a top-down-design bottom-up-implementation approach was used, where the major components that were required for the system to work were identified, and developed in isolation from one another. Once they were shown to work in isolation, the next step was to integrate these components into a complete system which could be used to achieve the goal of this project.

The algorithms developed in this project were tested incrementally using vehicle models of increasing complexity, starting with simple simulation models, then with more representative simulation models, and ultimately with the physical vehicles. The progression of vehicles that were used is shown in Figure 8.1, and became incrementally more representative of the real-world vehicles. The reason for using the different vehicles was to test the performance of the algorithms under different conditions, ranging from ideal simulation conditions to representative practical conditions.

Three kinds of vehicles were used, namely LiteSim vehicles, GazeboSim vehicles, and the real vehicles. The LiteSim vehicles are simple simulation models that are written in Python and implement the following dynamic model:

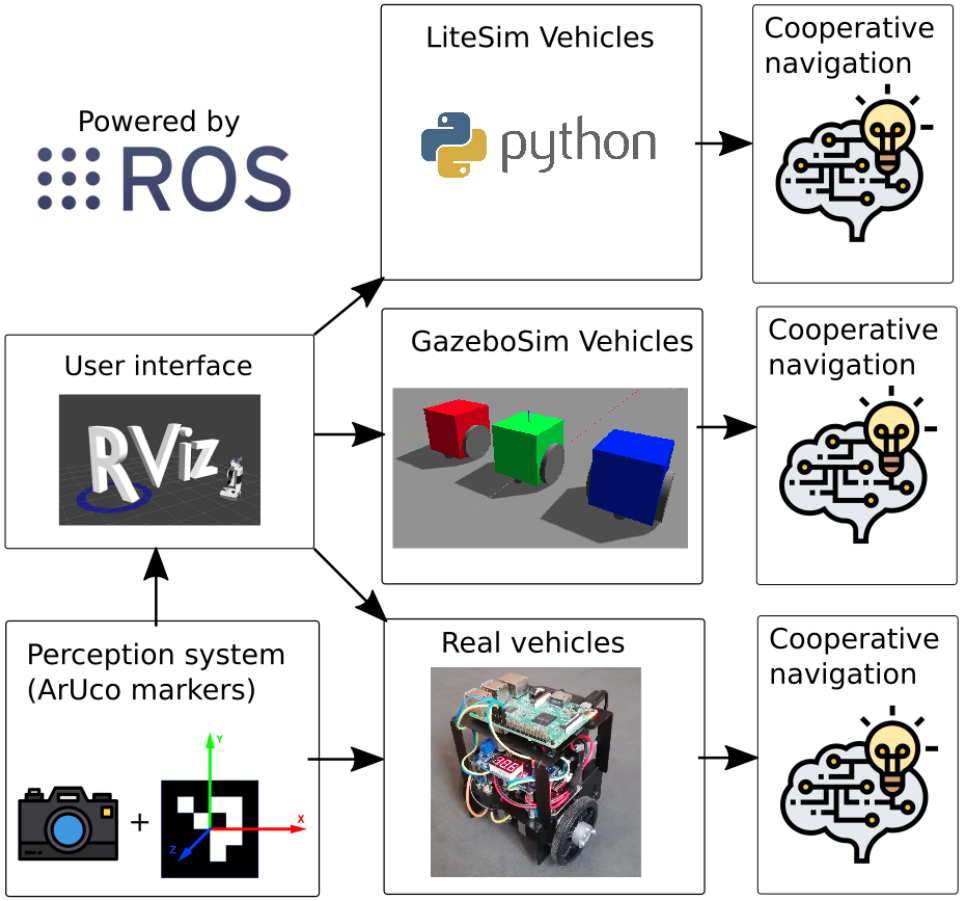


Figure 8.1: Several different kinds of vehicles were used to test the performance of the algorithms under different conditions. These vehicles vary in the extent to which they accurately model the real-world behaviour of vehicles.

$$\begin{pmatrix} x_{t+\Delta t} \\ y_{t+\Delta t} \\ \theta_{t+\Delta t} \end{pmatrix} = \begin{pmatrix} \Delta t \cos\theta_t & 0 \\ \Delta t \sin\theta_t & 0 \\ 0 & \Delta t \end{pmatrix} \begin{pmatrix} v_t \\ w_t \end{pmatrix} + \begin{pmatrix} x_t \\ y_t \\ \theta_t \end{pmatrix} \tag{8.1}$$

The GazeboSim vehicles implement more complex and representative models of the real vehicles, which include the inertia of the vehicles and the slipping between the wheels and the ground surface. The real vehicles are the three physical vehicles that are described in Chapter 6 and are used with the external pose estimation system described in Chapter 7 to perform the practical tests.

8.2. Overview of results

A video of the cooperative navigation system performing trajectory planning and execution in simulation and with the practical test setup and can be viewed online here (Viljoen 2020). Figure 8.2 shows the planned and executed trajectories for the LiteSim vehicles, GazeboSim vehicles, and physical vehicles respectively.

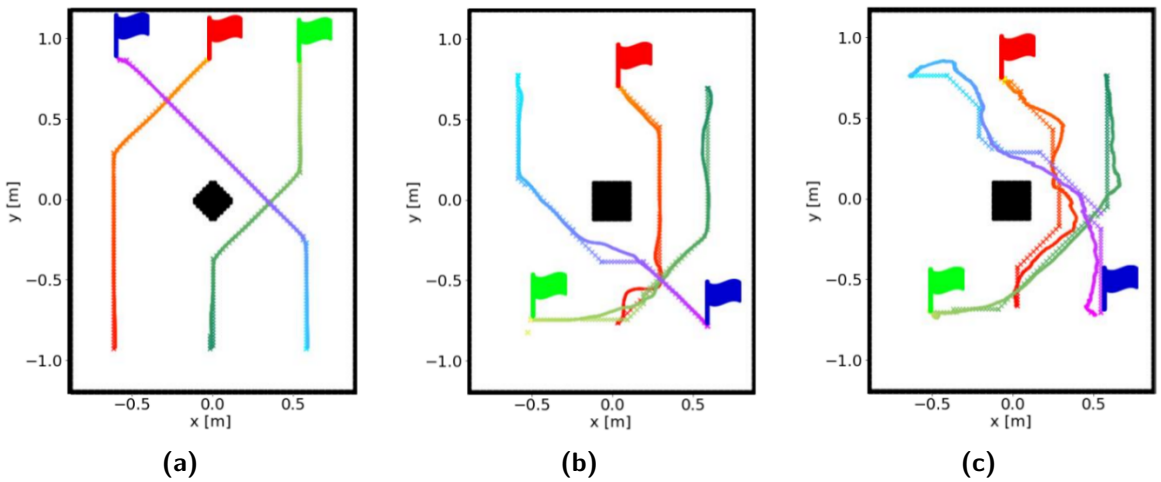


Figure 8.2: A comparison between the trajectory adherence for the three different kinds of vehicles that were used. In (a), the trajectory adherence of the LiteSim vehicles are shown, whereas that of the GazeboSim and practical vehicles are shown in (b) and (c) respectively.

During these tests, each vehicle's cooperative trajectory planning module finds and reserves collision-free trajectories for itself, taking into account the reserved trajectories of the other vehicles as well as the map of the static environment. The vehicles use a decentralised token allocation strategy for deciding when each vehicle is allowed to plan. Once the vehicles have found and reserved trajectories, they use their trajectory tracking modules to execute these trajectories. These trajectory tracking modules find the linear and angular velocity commands that the vehicles should execute to adhere to the reserved trajectories. The velocity commands are found iteratively for a receding time window using an MPC approach, and are then sent to the velocity controllers of the vehicles to be executed. The LiteSim vehicles execute the velocity commands by simply integrating them over time, while the GazeboSim and real vehicles execute the velocity commands by using their velocity controllers to actuate their differential drive motors. As expected, the adherence worsens as the testing goes from the idealised LiteSim vehicles to the higher fidelity GazeboSim simulation vehicles. This trend continues when moving from the GazeboSim vehicles to the real vehicles, as more non-idealities are introduced. Nonetheless, for both LiteSim and GazeboSim simulation vehicles, as well as for the real vehicles, the cooperative navigation algorithms are able to navigate the vehicles from their starting poses to their goal positions without any collisions occurring.

Figure 8.3 shows the trajectory deviation quantitatively for the LiteSim, GazeboSim and practical tests, as well as the optimisation time used for the three different tests. The average deviation during the LiteSim, GazeboSim and practical tests were 0.018 m, 0.022 m and 0.035 m respectively, with maximum deviations of 0.095 m, 0.171 m and 0.205 m. The deviation was calculated as the distance between where the vehicle was and where it was suppose to be for each time-step. This was found by performing four tests for each

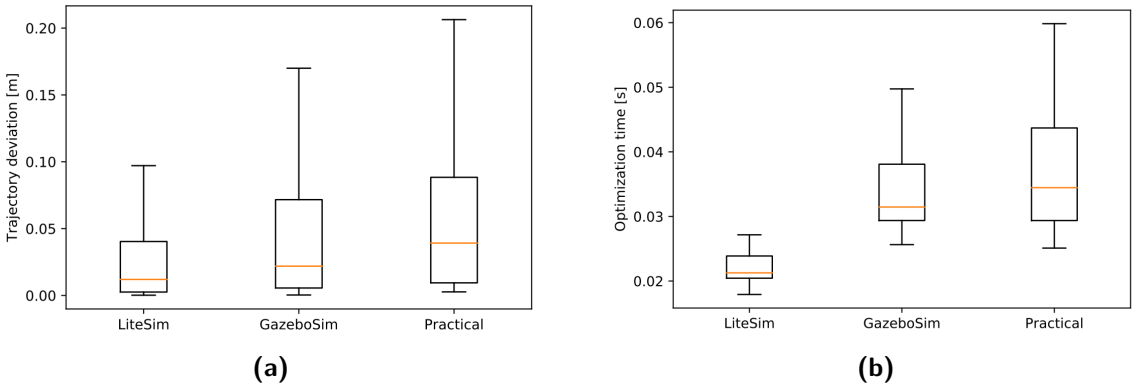


Figure 8.3: The quantitative results from the simulated and practical tests, showing the difference in trajectory deviation and optimisation time.

vehicle category, and using the deviation from all the vehicles in that category for each test. This data suggests that the increase in trajectory deviation that occur when vehicles which are incrementally more representative of real-world vehicles are used, results in a commensurate increase in optimisation time. This makes sense on an intuitive level, as the deviations results in more time having to be spent on finding a suitable trajectory that will allow the vehicle to rejoin the reserved trajectory.

The possibility of trajectory deviation was taken into account when designing the cooperative planning module, through the use of the vehicle footprint size parameter. This parameter defines the minimum distance between the vehicle and the nearest obstacle, taking into account the size of the vehicle as well as a margin for possible trajectory deviation. A vehicle footprint of 14 cm was used for the testing of the algorithms, which allowed for trajectory deviation of 8 cm, given that the radius of the vehicle is 6 cm. This deviation margin was sufficiently large to account for the majority of deviations when using both the simulated and physical vehicles. As can be seen from Figure 8.3, there were some deviations which exceeded the allowed margin, but these were seldom enough that no collisions occurred.

8.3. Cooperative trajectory planning evaluation

One of the essential parts of the cooperative navigation system is the ability of the vehicles to successfully find trajectories that are not in conflict with any of the other vehicles' trajectories. This ability is demonstrated in Figure 8.4, where three different examples of cooperative planning are shown. Three more such examples of cooperative trajectory planning can be found in Appendix B. The purpose of these tests is to demonstrate that the trajectory planner is able to plan collision-free trajectories for all of the vehicles in different scenarios and with different numbers of vehicles. In the first of these examples, only three vehicles are used, with the trajectories found in the following prioritised order: red, green, and then blue. The second example shows the same map being used, but this

time with six vehicles, with the order in which the trajectories are found being: red, green, blue, yellow, magenta and then finally cyan. From the space-time representation, it can be seen that using six vehicles instead of three results in a more cluttered space-time domain. However, valid trajectories could still be found for all six vehicles. The third example shows another scenario where six vehicles are used, with the order in which the trajectories

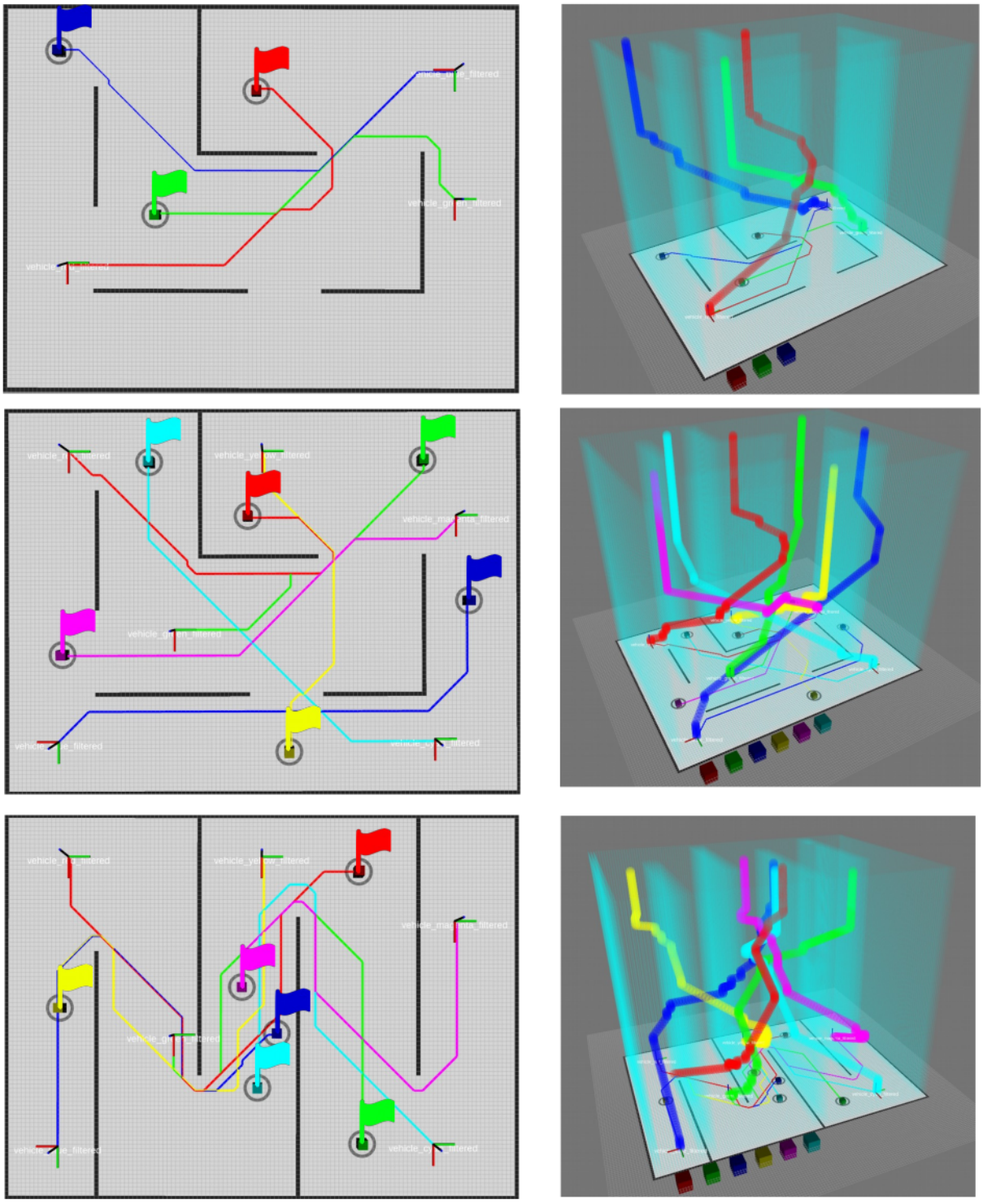


Figure 8.4: Three examples of cooperative trajectory planning, with the space-time representation shown on the right for each example.

are found being the same as in the second example. In this example, the vehicles are tasked with navigating past each other in a narrow corridor-like map. The result shows that all six the vehicles are able to find collision free trajectories, despite the narrowness of the corridor and the presence of the other vehicles. When using uncooperative planning techniques, these kinds of maps often result in deadlock situations, where the vehicles are unable to navigate past each other due to the bottleneck present in the map. The cooperative approach used here mitigates this problem, resulting in the reliable navigation of all six vehicles.

Planning time One of the important metrics that has to be considered when performing the cooperative planning, is the time taken by the vehicles to find the valid trajectories. This is important because the cooperative navigation must be performed in real time, where each vehicle periodically reserves a trajectory for itself. The vehicles perform the trajectory planning in an interleaved way, allowing each vehicle to keep its reserved trajectory updated. For this project, the time interval between each planning iteration was set to ten seconds, with no more than six vehicles used at a time. As all six of the vehicles had to complete their planning during each planning iteration, this meant that each vehicle could only use a maximum of 1.67 seconds, or a sixth of ten seconds. If any of the vehicles used more than this time, it could mean that one of the other vehicles might be unable to update its reserved trajectory, resulting in potential collisions. The planning time taken by the vehicles to find a trajectory was measured, and is shown in Figure 8.5, with most of the trajectories being found in under 0.2 seconds. The planning time was measured to verify that the trajectory planner can calculate solutions within the allowed planning time. Although there are two cases where the time taken to find a trajectory is significantly longer, it never exceeds 1.67 seconds.

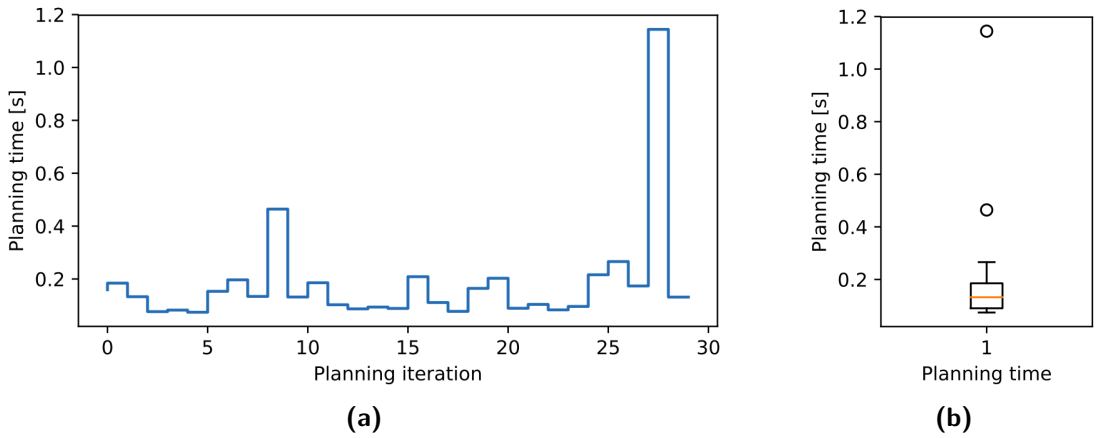


Figure 8.5: A sample of the length of time taken for the cooperative trajectory planner to find a valid solution is show in (a), with the distribution shown in (b).

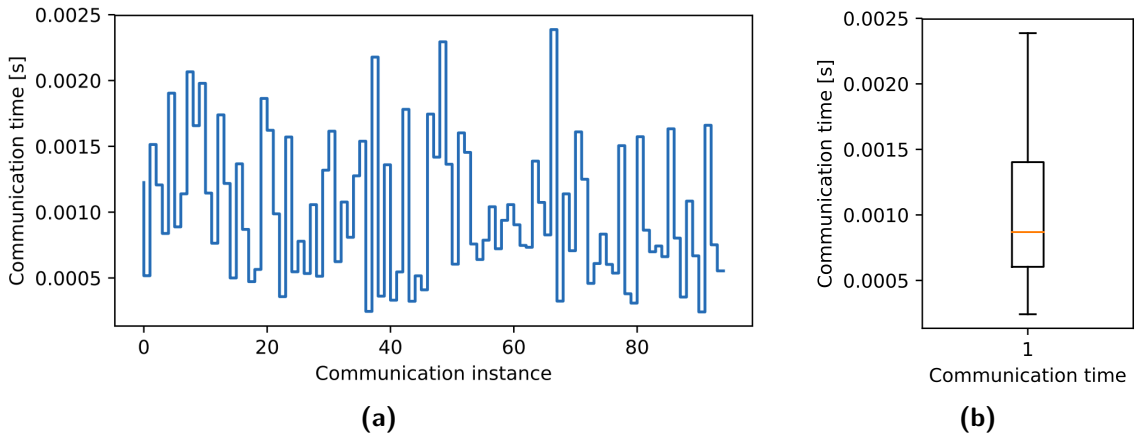


Figure 8.6: A sample of the time taken to communicate a trajectory to the other vehicles is shown in (a), with the distribution shown in (b).

Communication time One of the primary concerns for decentralised planning approaches is that it relies heavily on communication between the vehicles. This can cause severe problems when the communication between the vehicles is disrupted, either by being slowed down, or in the worst case being stopped completely. Providing error handling for communication errors or failures is beyond the scope of this project. However, it is still important to take the effect of communication delays into account when performing cooperative planning. The time taken to communicate between the vehicles was measured, and is shown in Figure 8.6, showing an average communication time of less than 1 millisecond. The communication delay is therefore negligible compared to the time taken to perform the trajectory planning. The communication delay also did not have a noticeable impact on the overall performance of the system.

8.4. Velocity controller evaluation

A box drive test was used to verify that the velocity controller actuates the vehicle to execute linear and angular velocity commands, and to determine how well the vehicle executes a reference trajectory in an open-loop fashion when the velocity commands for the trajectory are sent to the vehicle's velocity controller. This test consisted of commanding the vehicle to drive straight for five seconds at a linear speed of $0.2\,\mathrm{m\,s^{-1}}$, followed by a rotation speed of $18\,^{\circ}\,\mathrm{s^{-1}}$ for five seconds. This was repeated four times, so as to form a box, after which the measured path of the vehicle could be compared to the ideal box shaped response. This entire manoeuvre was repeated twice for each vehicle, in both the clockwise and anticlockwise directions, the results of which are shown in Figure 8.7. It is important to emphasise that the commands are executed in a purely open-loop fashion, with no feedback used to correct the vehicle's path. The more accurately the vehicle is able to execute the trajectory in an open-loop fashion, the easier it should be for the trajectory tracker to control the vehicle to follow the reference trajectory in a closed-loop fashion.

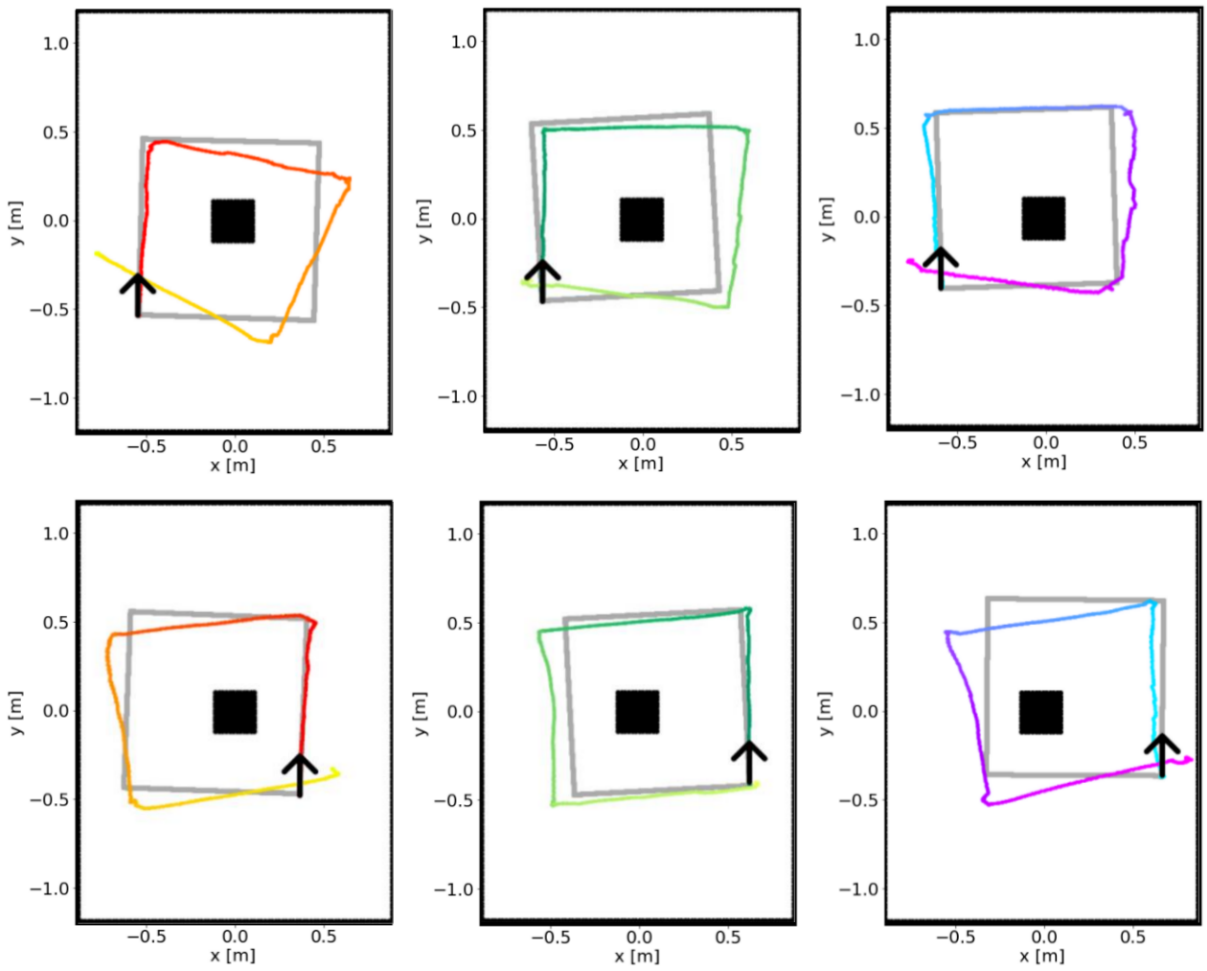


Figure 8.7: Results from the box drive test, with columns one through three for vehicles one through three respectively. Row one shows results from driving in the clockwise direction, whereas row two show the same but for the anticlockwise direction.

From these results it can be observed that the open loop accuracy for vehicle one performed the worst, with the largest degree of path deviation. An especially conspicuous deviation was present when driving in the clockwise direction with vehicle one, indicating that there is a problem with vehicle one's ability to perform clockwise turning. Vehicle two performed the best, with both the clockwise and anticlockwise manoeuvres showing a high degree of accuracy.

The time series plots of vehicle one's velocity, position, and yaw angle while performing the clockwise drive box test are shown in Figure 8.8. The plots show the commanded linear velocity and the commanded angular velocity. the vehicle's position and heading compared to the reference position and heading, and the position and heading tracking errors. The time-series plots of vehicle one's anticlockwise box drive test, as well as the other two vehicles' box drive tests (both clockwise and anticlockwise) can be found in Appendix C. Figure 8.8 shows a maximum heading deviation of 20°, with maximum deviations in the x and y directions of 25 cm and 35 cm respectively. The deviations in the x and y increase as the test progresses, with the maximum deviations in the x and y directions occurring

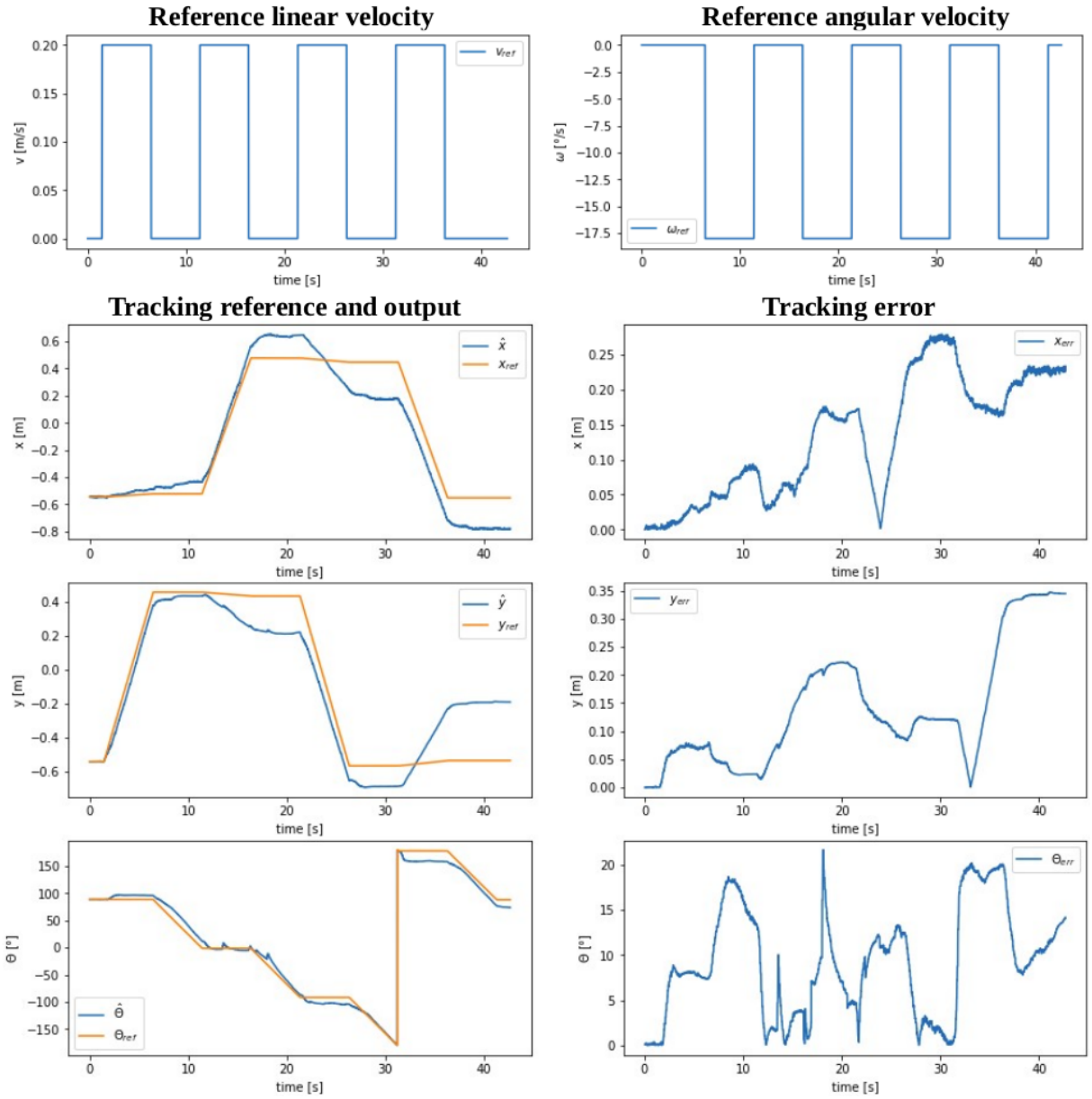


Figure 8.8: The time series plots recorded during vehicle one's box drive test in the clockwise direction.

after the vehicle has been controlled in an open-loop fashion for more than 30 seconds. The time-series plots highlights what was observed from the qualitative results, which is that the reference tracking start well, but then deviates as disturbances are introduced and the errors caused by model uncertainties compounds over time. This emphasises the need for a control technique that can account for these deviations, either through incorporating feedback or by applying the Model Predictive Control (MPC) approach used in this project.

8.5. Trajectory tracking evaluation

Once the vehicles were able to successfully find trajectories cooperatively in a decentralised way, the next step was to evaluate the ability of the vehicles to track those trajectories. This had to be done for all three the different classes of vehicles used, as they would differ in their ability to accurately track trajectories. For these tests the focus was not on the trajectory planning component, as this had already been tested. Rather the focus was on how accurately and reliably the vehicles could track the trajectories once they had been found. The results from evaluating the trajectory tracking ability of the three different classes of vehicles can be found in the following sections.

8.5.1. LiteSim complete system test

This section presents the results of the trajectory tracking tests that were performed using the LiteSim vehicles. The tests were performed by specifying goal locations for all three the vehicles, and having them cooperatively find trajectories that when executed will take them to their goal locations. A screenshot of the test is shown in Figure 8.9, with the goal locations of the vehicles indicated. Both the paths and space-time trajectories is shown for all the vehicles. Once they found the trajectories, the next step was to use their trajectory tracking modules to adhere as closely as possible to the trajectories. The video showing the footage from the tests is available on YouTube (Viljoen 2020).

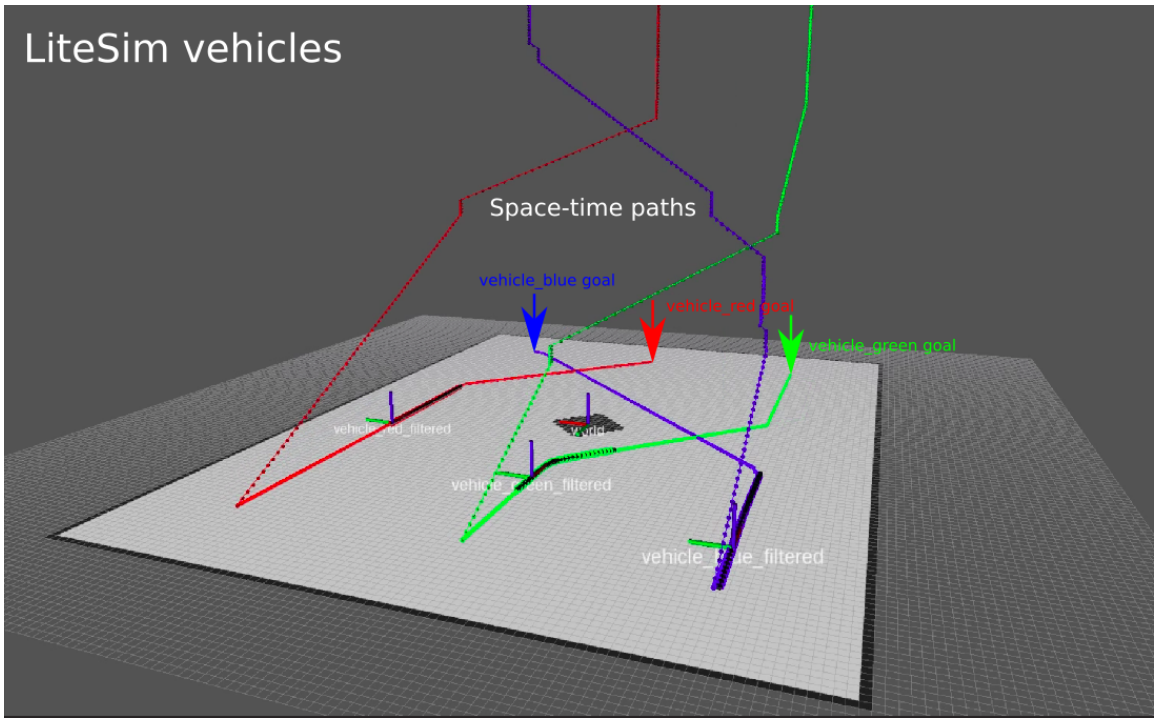


Figure 8.9: A frame showing the first test conducted, using the LiteSim vehicles. The vehicles were tasked with navigating to the indicated goal locations.

From Figure 8.10 it can be observed that there is little to no deviation from the reserved trajectory, resulting in collision free navigation. This can be seen more clearly in Figure 8.11, which shows both the reference pose and the estimated pose of the red vehicle for the duration of the test. Figure 8.11 also shows the error, which is calculated as the difference between the reference and estimated poses. From these plots it can be seen the vehicle exhibited accurate trajectory tracking capabilities, staying within 1 cm for most of the test. The large error spike in the heading of the vehicle is to be expected, as the cost function used when finding the optimised trajectory does not include heading adherence. The reason for this is that an error in the heading of the vehicle is sometimes required when recovering from a trajectory deviation. Similar plots for the green and blue vehicle can be found in Appendix D.

Figure 8.12 shows the time taken for the optimisation process during each planning iteration of the MPC. The time taken to calculate a solution never exceeded 35 ms for any of the vehicles. Given that the MPC loop executes at a frequency of 1 Hz, the optimisation time is well within the available time of 1 second. A total of 50 node points were used during the optimisation process for all the simulation and practical tests. These node points were spaced apart by intervals of 100 ms, resulting in a 5 second optimised trajectory.

Figure 8.13 shows both the linear and angular velocities executed by all three vehicles during the test. One interesting observation from these graphs is that there are two different levels of linear velocity, roughly $0.10\,\mathrm{m\,s^{-1}}$ and $0.14\,\mathrm{m\,s^{-1}}$. This is especially conspicuous for the linear velocity profile of the green vehicle. These two velocity levels are expected, as the same time is allocated for a diagonal move and a forwards move in the trajectory planning phase, even though the distances are different by a factor of $\sqrt{2}$.

Another interesting observation from the velocity graphs is that the linear velocity never exceeds $0.2\,\mathrm{m\,s^{-1}}$. This is again to be expected, as the trajectory optimisation specified that both the linear and angular velocities are constrained to the following ranges:

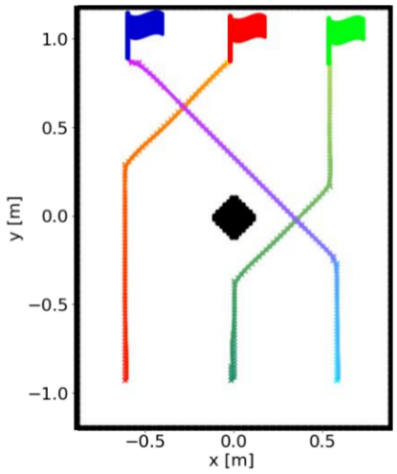


Figure 8.10: The trajectory adherence for the three vehicles during the LiteSim test.

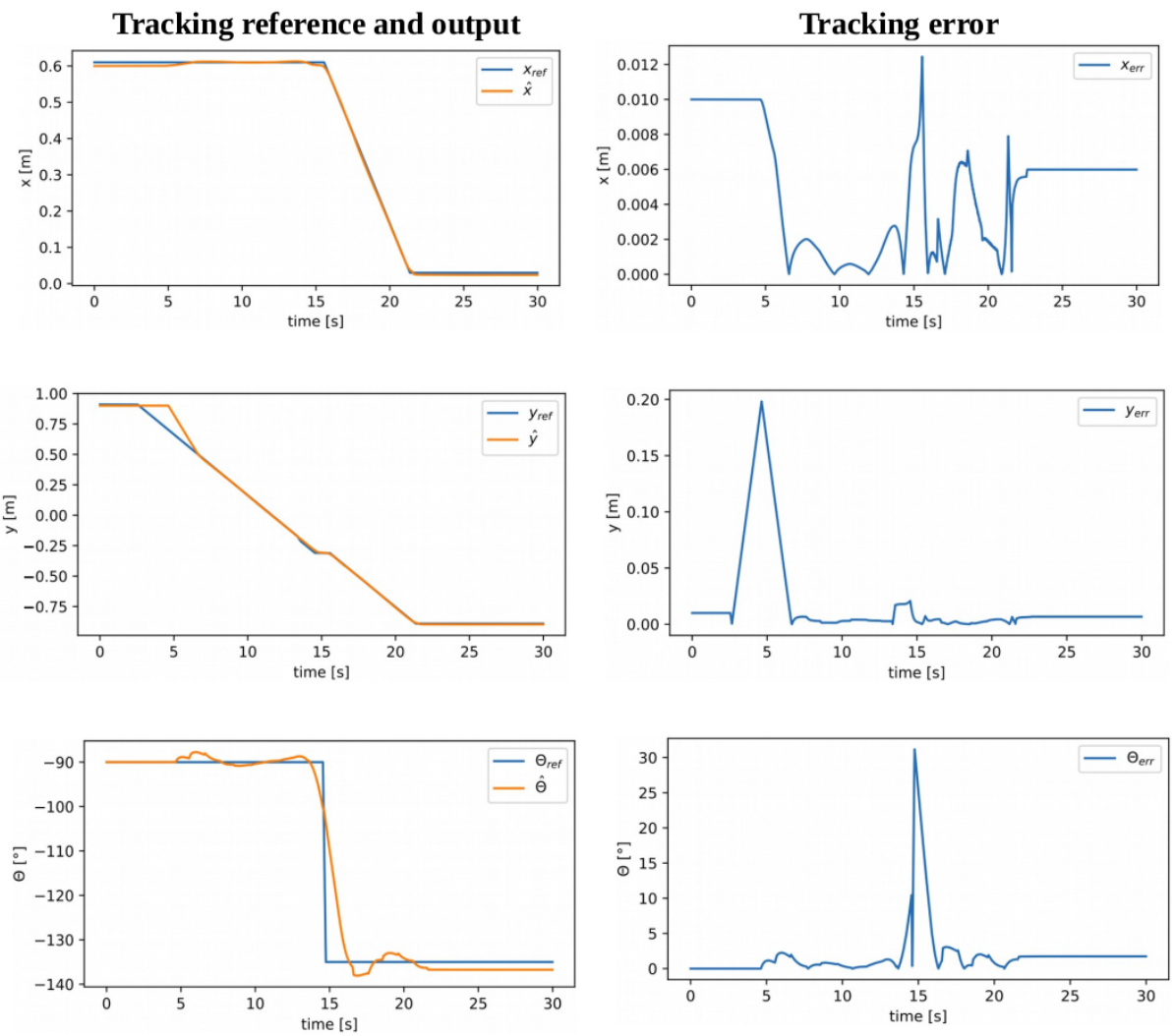


Figure 8.11: The plots of the reference and estimated values of the LiteSim red vehicle's state over time, as well as the error.

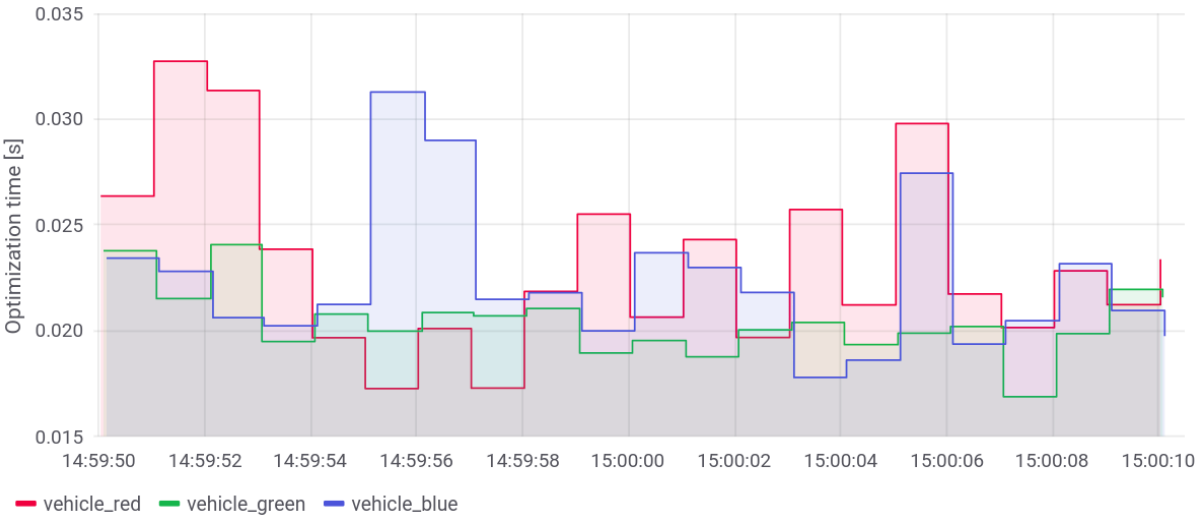


Figure 8.12: The trajectory tracking optimisation time for all three vehicles during the LiteSim test.

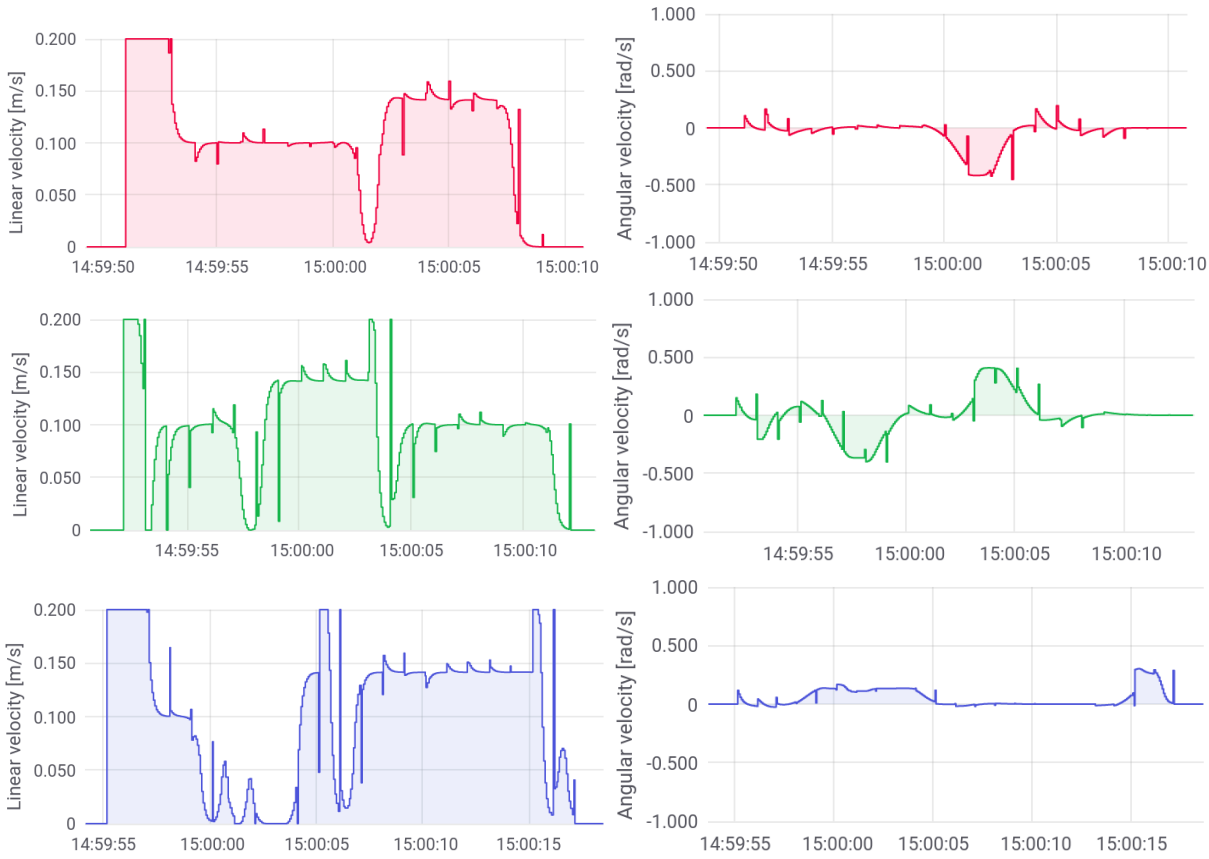


Figure 8.13: The velocities of all three vehicles during the LiteSim test.

$$\begin{aligned} v &\in [0, 0.2] \text{ m s}^{-1} \\ \omega &\in [-0.8, 0.8] \text{ rad s}^{-1} \end{aligned} \tag{8.2}$$

where v and ω are the linear and angular velocities of the vehicle.

This test was repeated three more times, with similar results. The adherence of the LiteSim vehicles for the other scenarios are shown in Figure 8.14, showing very little

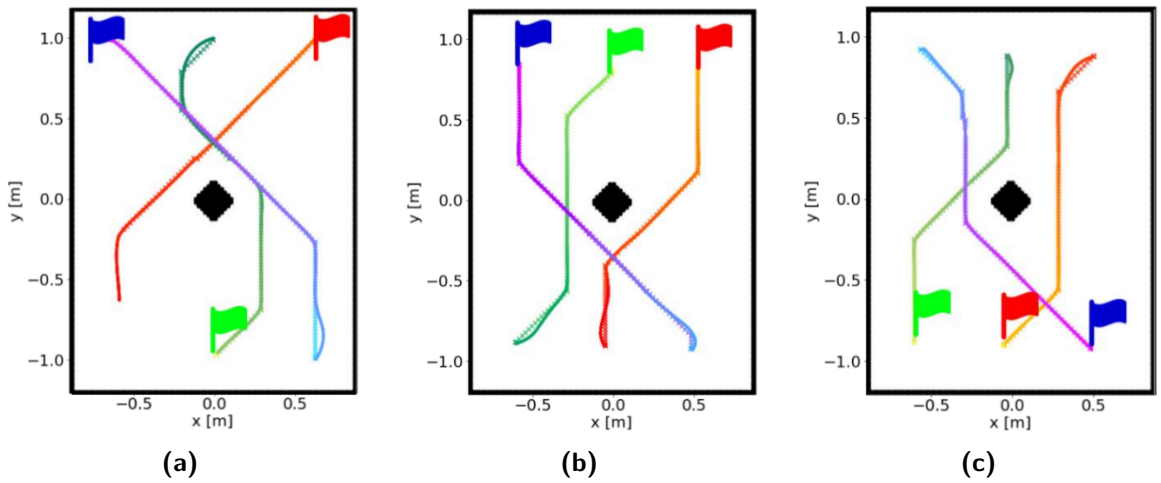


Figure 8.14: The trajectory adherence for three more scenarios using the LiteSim vehicles.

deviation from the reference trajectories.

8.5.2. GazeboSim complete system test

For the second phase of testing, the Gazebo simulation platform was used. Gazebo is a high fidelity simulation engine, capable of modelling the behaviour of various different kinds of robots. The model of the robot is specified using the Unified Robot Description Format (URDF), which allows for complex and flexible robot models. These URDF formats can also be visualised using the RViz visualisation platform, as shown in Figure 8.15. For this test, the same procedure was followed as before, with the goal locations of each of the vehicles being specified. The video footage of this test can also be found on YouTube (Viljoen 2020).

The resulting trajectory adherence in space-time can be seen qualitatively in Figure 8.16 for all three vehicles, with good adherence for the green vehicle, but less so for the blue and red vehicles. The red vehicle showed a transient oscillatory behaviour at the start of the trajectory execution, but displayed good adherence once this passed. This behaviour is not desired, but it does illustrate the ability of the vehicle to correct itself when deviating from the reference trajectory.

The qualitative results from the tests are shown in Figure 8.17, showing the adherence of the red vehicle. Similar plots are shown for the green and blue vehicles in Appendix D. From these quantitative results, it can be confirmed that the overall tracking ability of the

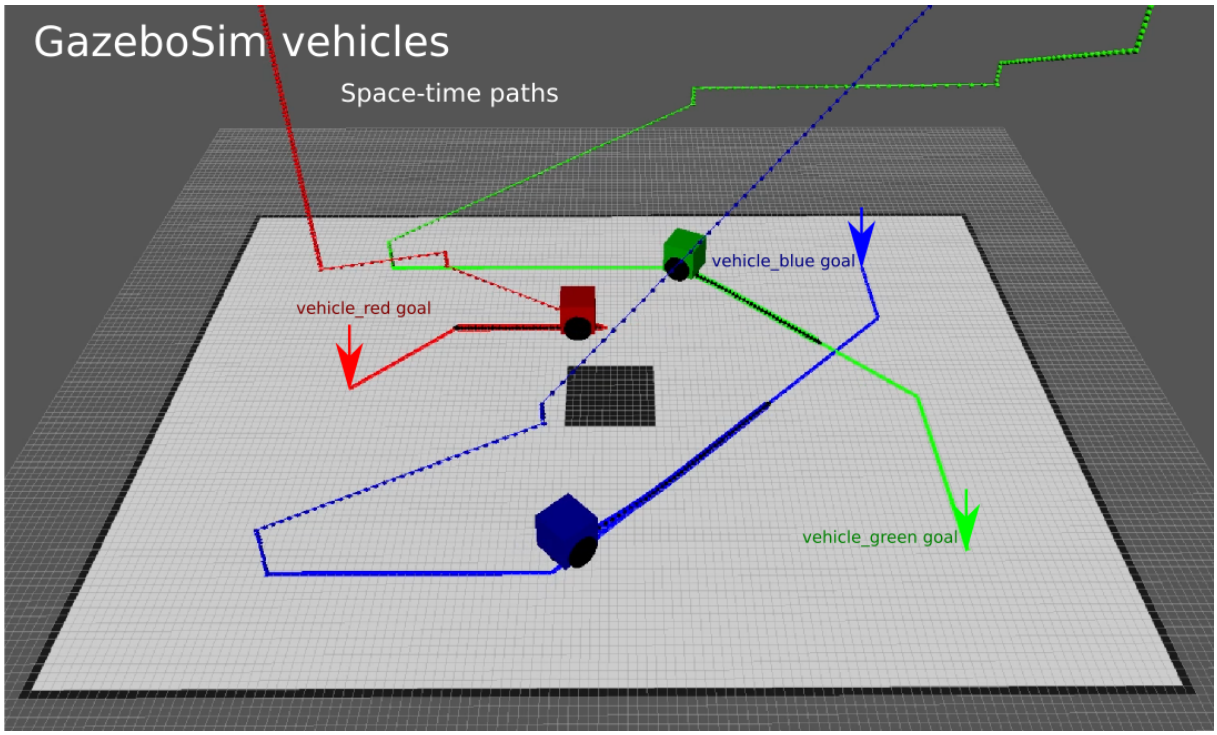


Figure 8.15: A frame showing the second test conducted, using the GazeboSim vehicles. The vehicles were tasked with navigating to the indicated goal locations.

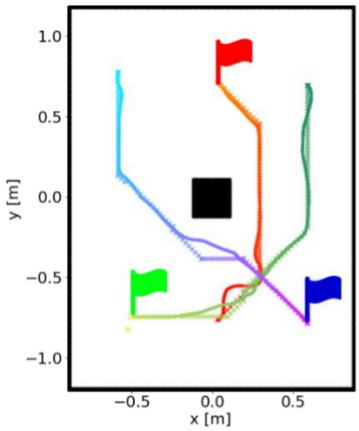


Figure 8.16: The trajectory adherence for the three vehicles during the GazeboSim test.

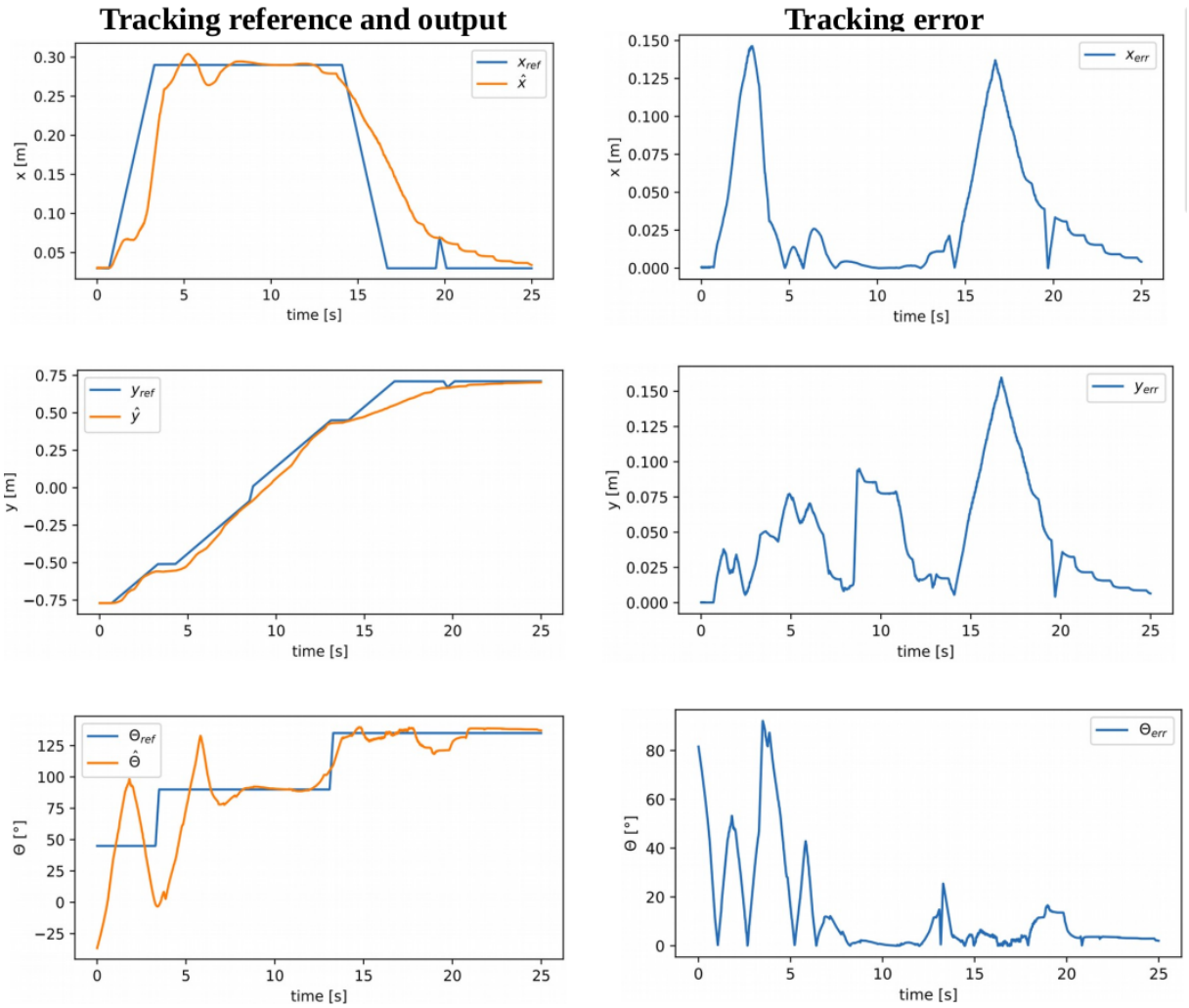


Figure 8.17: The plots of the reference and estimated values of the GazeboSim red vehicle's state over time, as well as the error.

GazeboSim vehicles is poorer than for the LiteSim vehicles, although for most of the tests the deviation was still less than 3 cm. For the red vehicle there is a noticeable deviation at the start of the test, caused by an overshoot in the angular control of the vehicle. This overshoot can be seen in Figure 8.17, as well as how the system reaches a steady-state

after roughly five seconds. There is a maximum heading deviation of more than 80° during this overshoot, resulting in deviations in the x and y direction of 0.15 meter and 0.075 m respectively. Once the steady-state is reached, the trajectory tracking improves significantly. This suggests that the controller behaves more optimally once the vehicle is in motion, being more prone to deviations at the start of the trajectory execution.

Figure 8.18 shows the velocity profiles of the three vehicles. Two notable differences are observed in the velocity profiles of the GazeboSim vehicles compared to those of the LiteSim vehicles. The first difference is that the GazeboSim vehicles all exhibit higher commanded velocities than the LiteSim vehicles. There are large sections where the linear velocity is saturated, indicating that the vehicles were lagging behind, and had to increase their velocities to "catch up" to where they were supposed to be. The second difference is that the velocity profiles for the GazeboSim vehicles exhibit spikes that occur roughly every second, just after every trajectory optimisation iteration. The reason for these spikes is that the trajectory optimiser places a large weighting on the space-time adherence of the vehicles, and uses these bursts of speed to force the vehicles back onto the reserved path. Unfortunately this can often have an adverse effect, as the sudden burst of speed can result in a shaky behaviour which can cause larger deviations.

The time-series plots of the optimisation times for the GazeboSim vehicles is shown in Figure 8.19. The average optimisation time is slightly longer for the GazeboSim vehicles

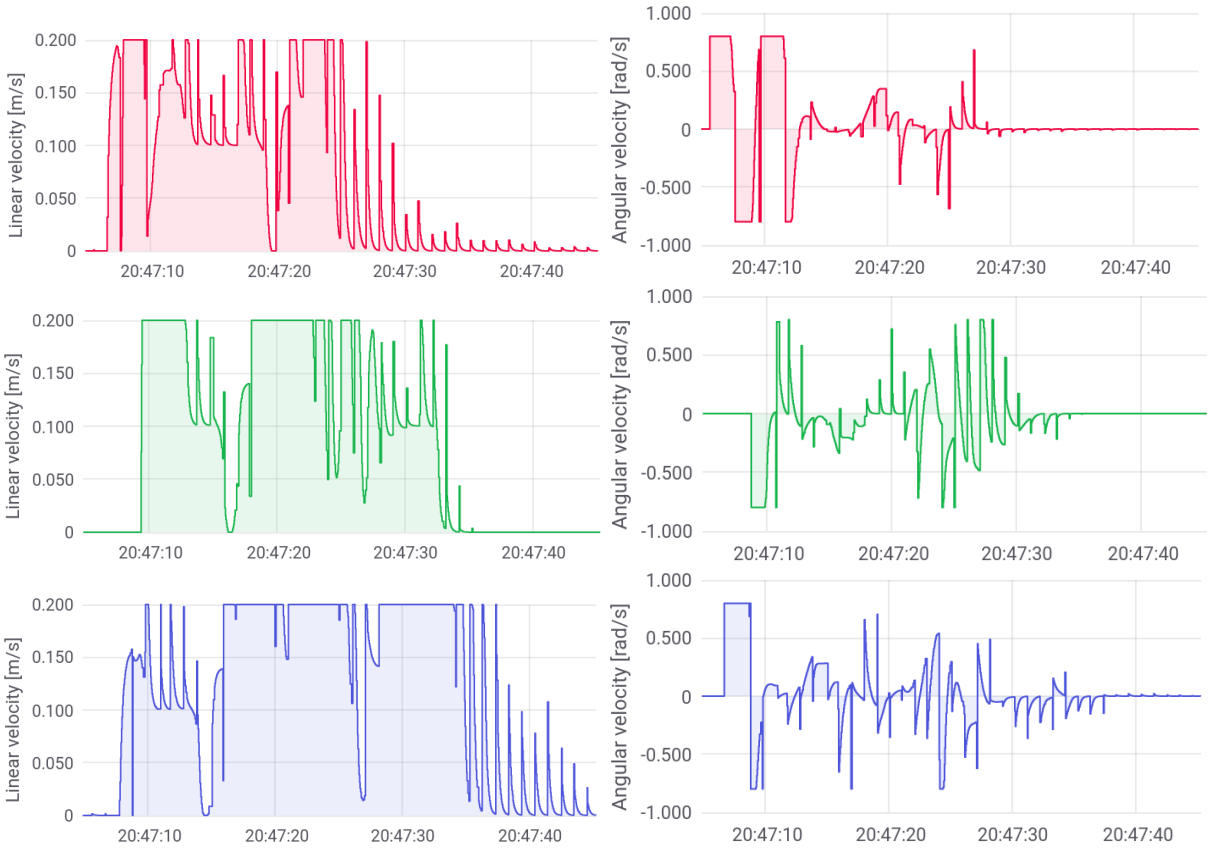


Figure 8.18: The velocities of the three vehicles during the GazeboSim test.

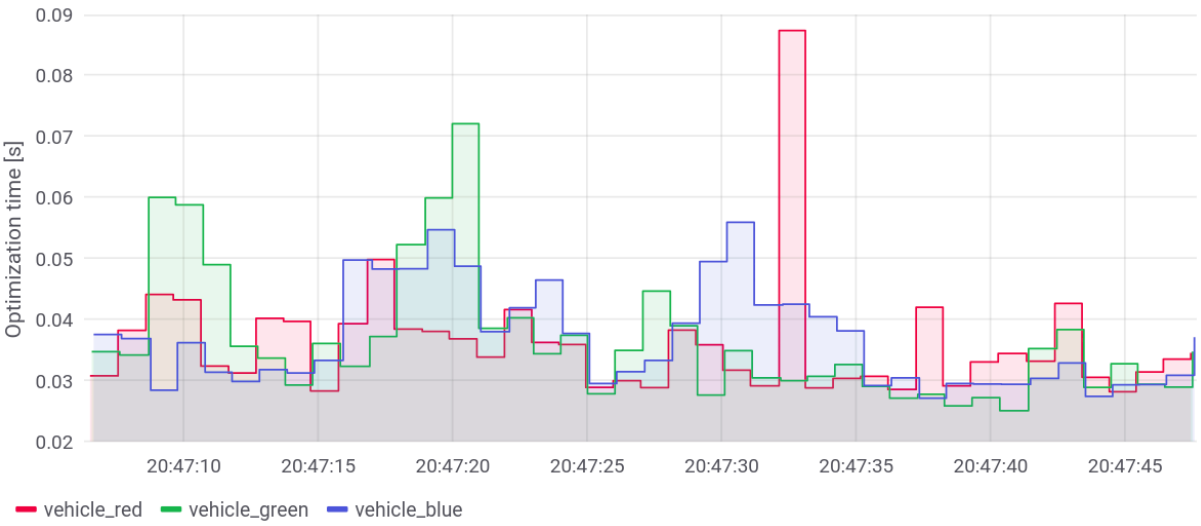


Figure 8.19: The trajectory tracking optimisation time for the three vehicles during the GazeboSim test.

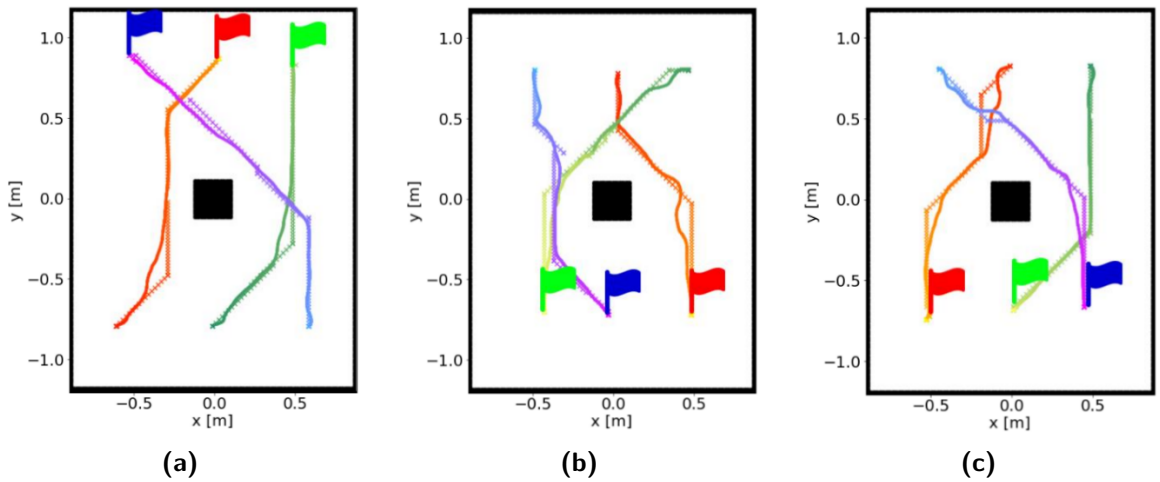


Figure 8.20: The trajectory adherence for three more scenarios using the GazeboSim vehicles.

than for the LiteSim vehicles. However, the optimisation time is still well within the 1 second that is allowed. Three more tests were conducted, the planned and executed trajectories are shown in Figure 8.20. Even though there are sections where the vehicles deviates from their trajectories, the overall adherence is still good, with no collisions occurring.

8.5.3. Practical complete system test

The final phase of the testing was to evaluate the cooperative navigation system using the physical vehicles and the external pose estimation system. Figure 8.21 shows a frame from this test, with the paths and space-time trajectories of all three the vehicles shown. The footage of both the cameras used to localise the vehicles can also be seen in Figure 8.22. The complete footage for this test can be found on YouTube (Viljoen 2020).

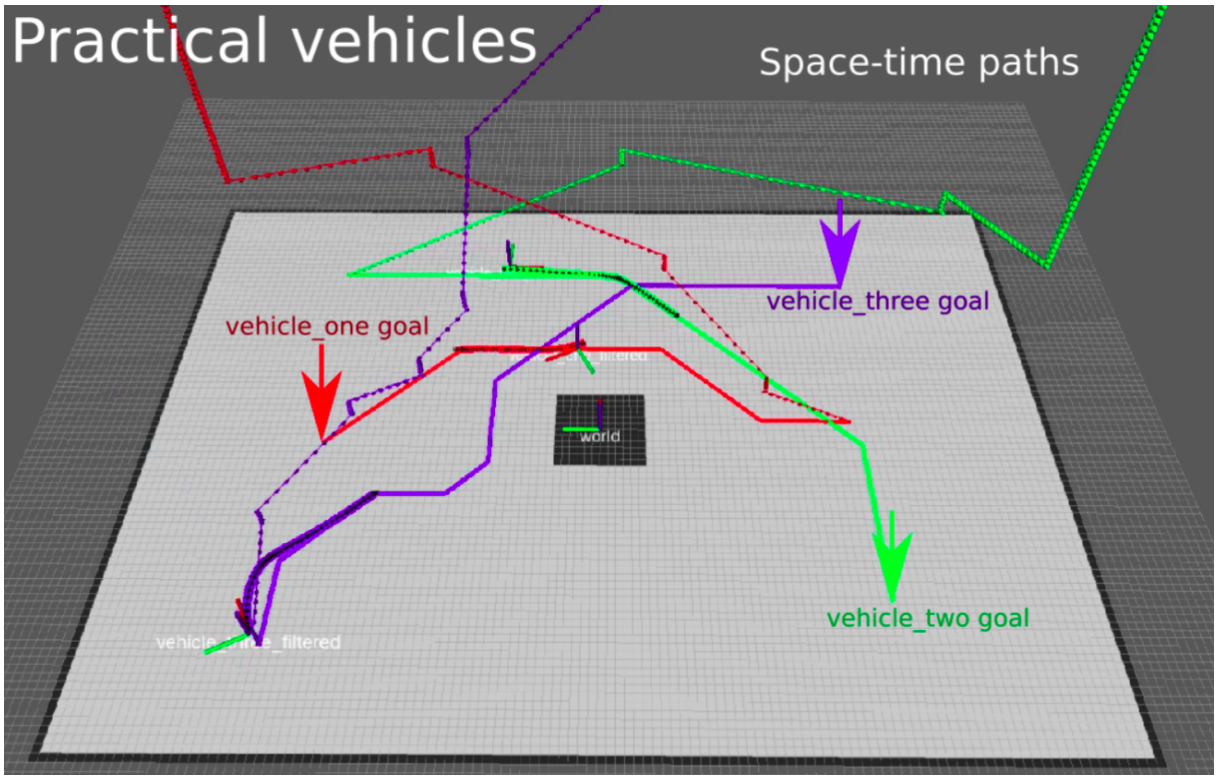


Figure 8.21: A frame showing the third test conducted, using the practical vehicles. The vehicles were tasked with navigating to the indicated goal locations.

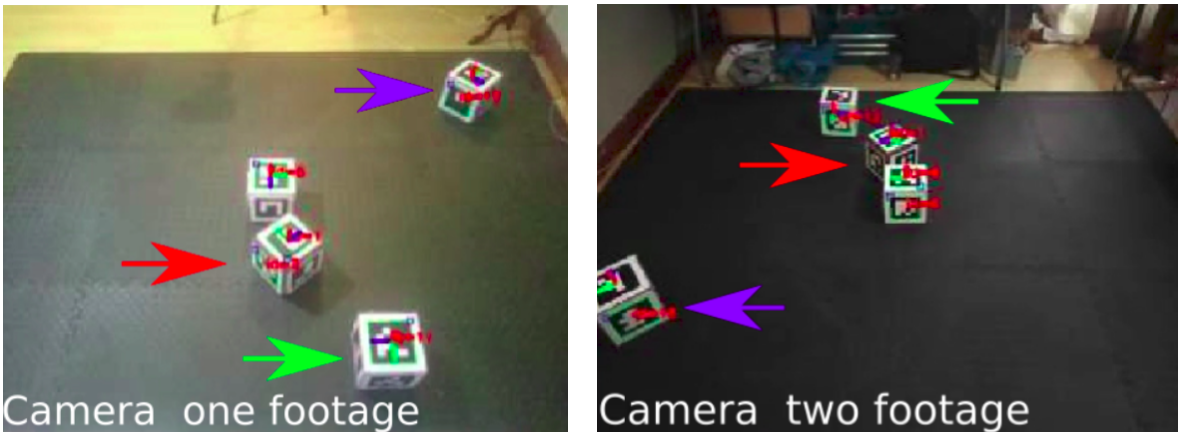


Figure 8.22: The footage from the two cameras used during the practical test, with the positions of vehicles one, two and three indicated by the red, green and blue arrows.

Figure 8.23 shows the trajectory adherence of the vehicles, with noticeably more deviation than either of the simulated tests. This was expected, as the practical tests introduced a number of non-idealities that were not modelled during the simulated tests, such as the wheels being slightly misaligned. Overall there was still acceptable adherence, with no collisions occurring. The time series plots of the velocity commands, reference and measured position trajectory, and reference and measured heading of vehicle one is shown in Figure 8.24. The time series plots for the other two vehicles can be found in Appendix D.

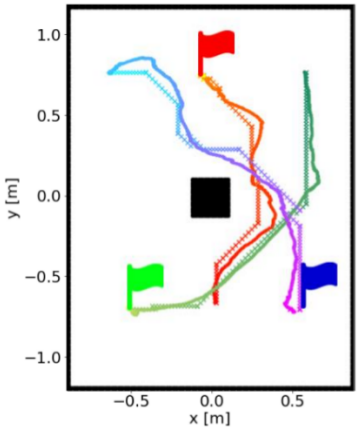


Figure 8.23: The trajectory adherence for the three vehicles during the practical test.

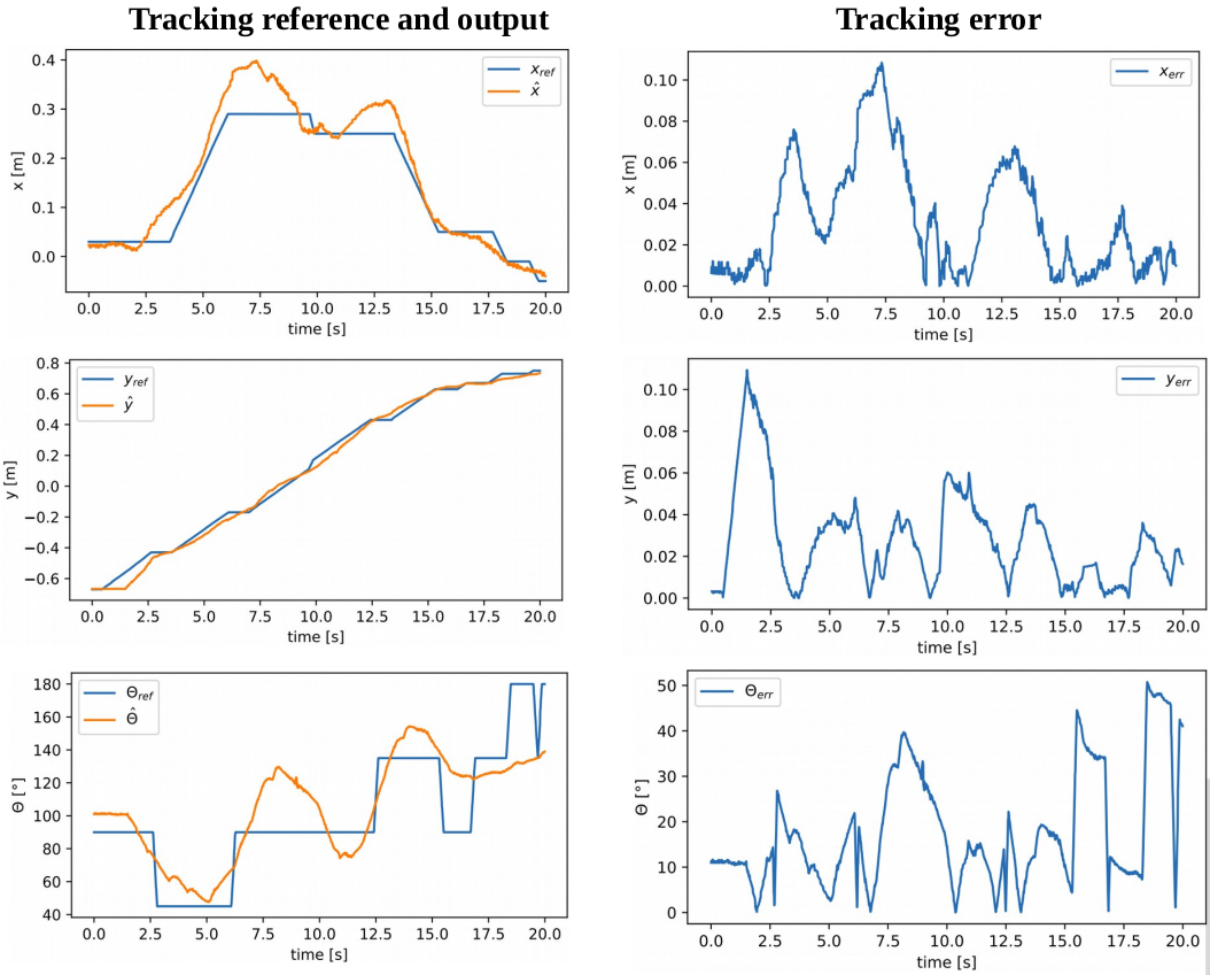


Figure 8.24: The plots of the reference and estimated values for the practical test red vehicle's state over time, as well as the error.

The velocity profiles of the vehicles are shown in Figure 8.25, with similar behaviour as for the GazeboSim test, again showing large sections where the velocity saturates against the imposed limits. The optimisation times can be seen in Figure 8.26, which was consistently less than 0.1 s. Figure 8.27 shows three more test that were conducted using the practical vehicles. One interesting observation is that for the test in Figure 8.27 (c),

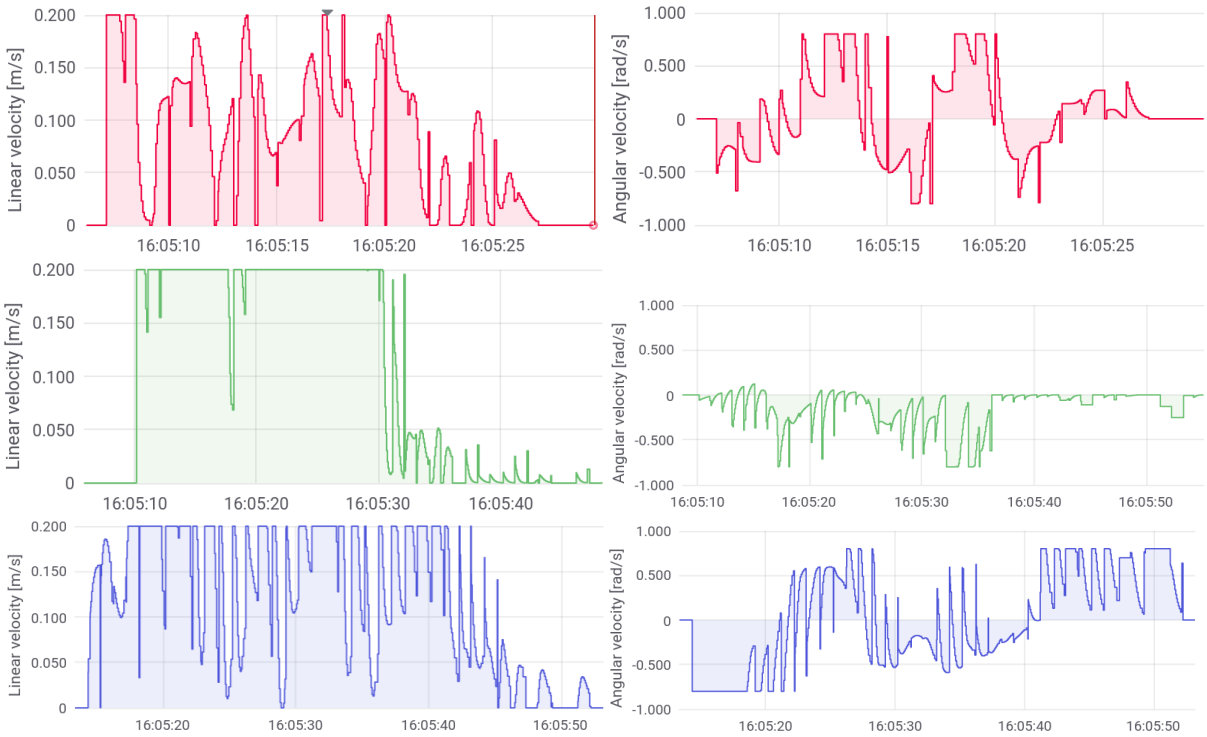


Figure 8.25: The velocities of the three vehicles during the practical test.

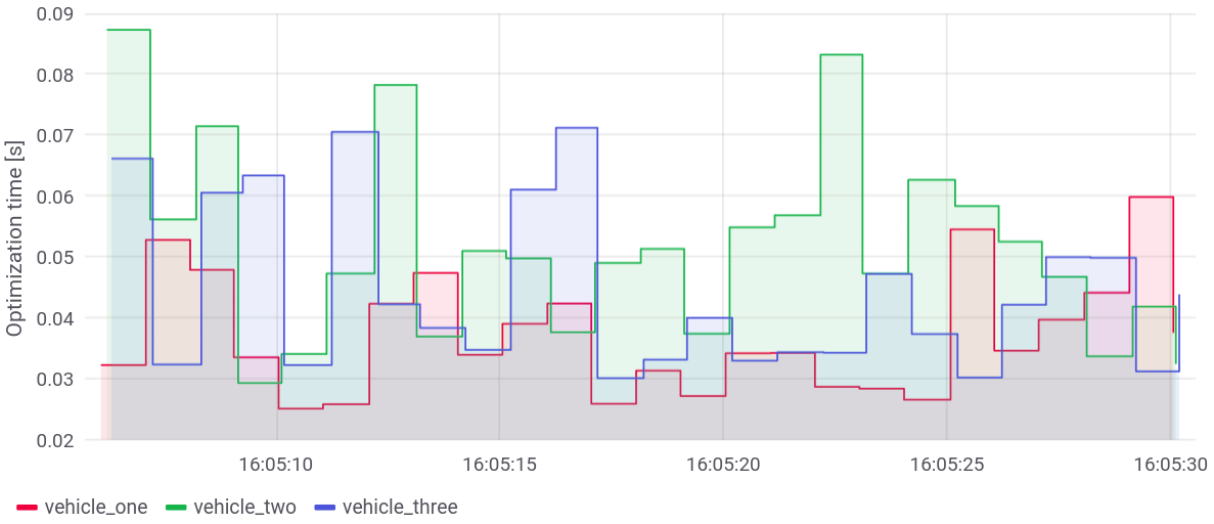


Figure 8.26: The optimisation time for the three vehicles during the practical test.

the deviation of the green vehicle is corrected by the trajectory replanning that occurs. The high level cooperative trajectory planner replans at a fixed frequency of 0.1 Hz unless a new goal is specified, in which case it replans immediately. This replanning allows the vehicle to correct for deviations that might occur, as well as taking into account changes in the reserved trajectories of the other vehicles.

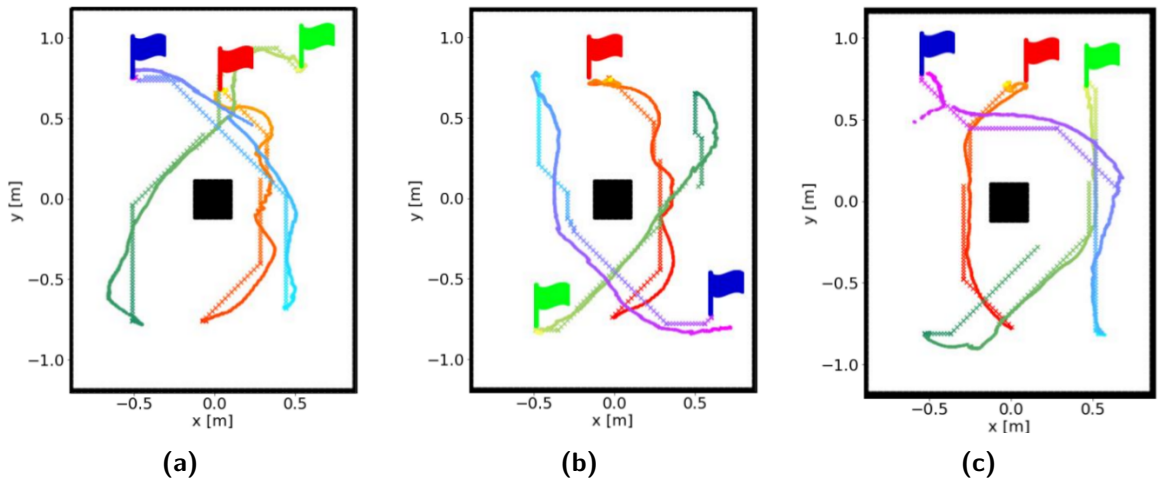


Figure 8.27: The trajectory adherence for three more scenarios using the practical vehicles.

8.6. Conclusion

This chapter described the tests that were performed to evaluate the performance of the cooperative navigation system in simulation and with practical tests. This was done by first testing the individual subsystems, after which the performance of the complete system was evaluated when deployed on three different classes of vehicles. The system was able to successfully navigate several AGVs both in simulation and using a practical setup, without causing collisions between the vehicles or with the static environment, and using a decentralised coordination and communication technique. As expected, the performance of the system was worse when using physical vehicles as opposed to simulated ones.

Chapter 9

Conclusion and Recommendations

9.1. Summary of work

The purpose of this project was to design a cooperative navigation system that can navigate multiple Autonomous Ground Vehicles (AGVs) within an environment using cooperative trajectory planning and execution. An overall system architecture was designed that includes the following modular components: a cooperative trajectory planner, a trajectory tracker, and a velocity controller.

A cooperative trajectory planning module was developed to plan trajectories for the vehicles to navigate from their initial positions to their goal positions while avoiding collisions with one another and with static obstacles in the environment. The planner takes the kinematic constraints of the vehicles into account when planning the trajectories. The Windowed Hierarchical Cooperative A* (WHCA*) multi-agent planning algorithm was used to plan collision-free trajectories that minimise the total distance travelled for each vehicle. The vehicle trajectories are planned sequentially using a pre-determined priority order, and the planned trajectories are saved in a each of the vehicles' reservation tables.

A trajectory tracking module was developed to execute the planned trajectories that are provided by the trajectory planning module. A separate trajectory tracking node is implemented for each vehicle. The trajectory tracking module listens for the planned trajectories published by the cooperative planning module and uses a Model Predictive Controller (MPC) to calculate the velocity commands for the vehicle to adhere to its planned trajectory. The trajectory tracking controller uses feedback control to correct deviations of the vehicle's actual trajectory from its planned trajectory, and takes the static environment and the kinematic constraints of the vehicle into account to calculate velocity commands that provide collision-free trajectories to rejoin the planned trajectory.

A velocity controller module was developed to control the vehicle to execute the linear and angular velocity commands provided by the trajectory tracking module. The velocity controller translates the linear and angular velocity commands into common and differential stepper motor commands for the vehicle's differential drive system. The vehicles' linear and angular velocities were therefore controlled in an open-loop fashion.

The integrated cooperative navigation system was tested both in simulation and with practical experiments using physical vehicles. Two different simulated vehicle models were implemented, called LiteSim and GazeboSim vehicles respectively. The LiteSim vehicles are lightweight simulation models that were written in Python and implement only the first-order vehicles kinematics. The GazeboSim vehicles are more complex and more representative of the actual vehicles and were developed in the Gazebo simulation environment. For the practical experiments, three physical vehicles were designed and built. The physical vehicles used Raspberry Pi single-board computers as their onboard computer and implemented a differential drive system consisting of two stepper motors with stepper motor drivers. An external vision-based pose estimation system was developed to determine the poses of the physical vehicles in real time. The vehicle poses are required by both the trajectory planning and the trajectory tracking modules. The pose estimation was accomplished by placing ArUco fiducial markers on the vehicles and using external cameras and computer vision algorithms to detect the marker and determine the vehicles poses.

The trajectory planning module and trajectory tracking modules were implemented as ROS nodes on a central ground station computer, while the velocity controller modules were implemented as ROS nodes on the onboard computers of the physical vehicles. The communication between the ground station computer and the vehicles was performed using WiFi communications.

The results of the simulation tests and the practical tests showed that the cooperative navigation system is able to successfully plan and execute trajectories for multiple ground vehicles to navigate an environment containing static obstacles. The trajectory planning module is able to plan collision-free trajectories for multiple vehicles. The trajectory tracking module is able to control the vehicles to follow their planned trajectories with an average trajectory deviation of less than 4 cm, despite the non-ideal open-loop velocity execution of the physical vehicles. This trajectory tracking accuracy is acceptable given that the tracking module plans using a 12 cm safety margin for collision avoidance clearance.

The timing measurements taken during the simulation and practical tests indicate that the cooperative navigation system should be viable for real-time implementation. The trajectory planning module plans the trajectories with an average algorithm execution time of 0.2 seconds, and a maximum execution time of 1.2 seconds, which is well within the allowed planning interval of 10 seconds. The trajectory tracking module executes the model predictive controller with an average algorithm execution time of 0.036 seconds and a maximum execution time of 0.059 seconds, which is within the sampling interval of 1 second.

9.2. Recommendations

The following recommendations are made for future research:

General improvements

- The cooperative navigation system currently only provides collision avoidance for cooperative vehicles and static obstacles in the environment, and does not provide collision avoidance for uncooperative dynamic obstacles. Adding the functionality that allows for navigating in the presence of uncooperative agents would increase the usefulness and applicability of the algorithms
- The trajectory planning and tracking modules have only been demonstrated using the kinematic constraints for differential drive vehicles. The cooperative navigation system should also be demonstrated using the kinematic constraints of other types of vehicles, such as Ackermann vehicles.
- The software for the cooperative navigation system could be migrated from ROS 1 to ROS 2. The reason for rather using ROS 2 is because ROS 1 uses a centralised *roscore*, which introduces a single point of failure. Although the planning and tracking modules can be used in a decentralised way, the use of ROS 1 prevents the system from being fully decentralised. The system can be made fully decentralised by implementing it in ROS 2. The reason ROS 1 was used for this project was because at that time ROS 2 was still in a less mature state. While still being in active development, ROS 2 is now much more usable and well documented.

Cooperative trajectory planner

- One limitation of the cooperative planner is that it is unable to vary the vehicle's speed, always translating and rotating at fixed speeds. By accommodating various speeds during the planning phase, it would be able to find more optimal solutions.
- Anytime planning could be implemented, which would allow the planner to always return a trajectory within a predetermined time window, or return an error if no valid trajectory could be found for that time window. This would solve the problem where one vehicle's planning takes too long, and prevents the other vehicles from planning.
- The key principle used to facilitate the cooperative navigation of the vehicles is that they are able to reserve trajectories, which are then avoided by the other vehicles. It is therefore a problem if the vehicles do not adhere to these reserved trajectories, as this increases the likelihood of collisions. One way of mitigating this problem is

by automatically triggering a replanning when a vehicles deviates from its reserved trajectory by more than a predefined margin. Adding this feature will improve the resiliency and robustness of the overall system.

- One of the main features of the cooperative trajectory planner is that it can facilitate decentralised planning. It does this by using a token allocation strategy, whereby the agents plan sequentially according to their predefined priorities. During the testing of the algorithms, the token allocation would sometimes fail, allowing two vehicles to plan simultaneously. This was most likely caused by timing issues, when two vehicles decide to start planning at precisely the same time. This could result in collisions, as the vehicles are then unable to take each other's reserved trajectories into account. A recommendation would be to redesign the token allocation strategy to find a more robust way of deciding which vehicles plans when.

- As the communication and coordination between the vehicles is the enabling mechanism for the decentralised planning, network latency could severely degrade the overall performance of the system. Further research could be done to find robust mechanisms for handling network delays, as well as to optimise the system for minimum communication delay between vehicles.

Trajectory tracker

- A significant improvement to the current system could be achieved by extending the trajectory tracker with the ability to take the position of the other vehicles into account when finding the optimised trajectory. This is necessary because sometimes the vehicles deviate from their reserved trajectories, potentially obstructing the trajectories of the other vehicles. The trajectory tracking module currently has the ability to accommodate only the static obstacles when optimising the trajectory, to prevent any collisions with them when recovering from deviations. This would have to be extended to include the other vehicles, predicting their most likely future positions based on where they are at that moment as well as their reserved trajectories. Adding this feature would result in a more robust and reliable operation of the overall system.

- The trajectory tracking module could be further improved by modelling the state of the vehicles using second order models, to calculate the suitable velocities as well as accelerations that the vehicles should execute to track the reserved trajectories. The current implementation disregards the acceleration of the vehicles, and could output velocity commands which cause the vehicles to exhibit non-ideal behaviour. This non-ideal behaviour is caused by spikes in the commanded velocities, which can be avoided by limiting the allowed acceleration of the vehicles.

Practical test environment

- The localisation of the vehicles can be improved by using a more advanced filtering technique, such as the Kalman Filter. By improving the localisation accuracy of the vehicles, the overall performance of the system would be improved as well.
- A vehicle fleet management system could be developed which can be used to monitor the status of the vehicles, such as the remaining battery life and computational performance metrics. This could also be used for managing over-the-air software upgrades for the vehicles, such as uploading the latest algorithms as they are developed. As this project only made use of three vehicles at a time, this was not deemed necessary, but it would become essential if a large number of vehicles are used.

Bibliography

Amidi, O. and Thorpe, C.E. (1991 mar). Integrated mobile robot control. In: *Mobile Robots V*, vol. 1388, pp. 504–523. SPIE.

Andersson, J.A., Gillis, J., Horn, G., Rawlings, J.B. and Diehl, M. (2019 mar). CasADi: a software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, vol. 11, no. 1, pp. 1–36. ISSN 18672957.

Arena, F. and Pau, G. (2019 jan). An Overview of Vehicular Communications. *Future Internet*, vol. 11, no. 2, p. 27. ISSN 1999-5903.

ArUco (2020). ArUco: a minimal library for Augmented Reality applications based on OpenCV — Aplicaciones de la Visión Artificial.
Available at: https://www.uco.es/investiga/grupos/ava/node/26

Barraquand, J. and Latombe, J.C. (1991). Robot Motion Planning: A Distributed Representation Approach. *The International Journal of Robotics Research*, vol. 10, no. 6, pp. 628–649. ISSN 17413176.

Behmanesh, M., Hong, T.S., Kassim, M.S.M., Azim, A. and Dashtizadeh, Z. (2017 may). A brief survey on agricultural robots. *International Journal of Mechanical Engineering and Robotics Research*, vol. 6, no. 3, pp. 178–182. ISSN 22780149.

Boddy, M. and Dean, T. (1989). Solving Time-Dependent Planning Problems. Tech. Rep..

Borges, M., Symington, A., Coltin, B., Smith, T. and Ventura, R. (2018 dec). HTC Vive: Analysis and Accuracy Improvement. In: *IEEE International Conference on Intelligent Robots and Systems*, pp. 2610–2615. Institute of Electrical and Electronics Engineers Inc. ISBN 9781538680940. ISSN 21530866.

Botea, A., Botea, A., Müller, M. and Schaeffer, J. (2004). Near optimal hierarchical path-finding. *JOURNAL OF GAME DEVELOPMENT*, vol. 1, pp. 7—-28.

Cadena, C., Carlone, L., Carrillo, H., Latif, Y., Scaramuzza, D., Neira, J., Reid, I. and Leonard, J.J. (2016 jun). Past, Present, and Future of Simultaneous Localization And Mapping: Towards the Robust-Perception Age. *IEEE Transactions on Robotics*, vol. 32, no. 6, pp. 1309–1332. 1606.05830.

Capek, K. (2020). Karel Capek - Who did actually invent the word "robot" and what does it mean?
Available at: https://www.npr.org/

Chand, B.N., Mahalakshmi, P. and Naidu, V.P. (2018 feb). Sense and avoid technology in unmanned aerial vehicles: A review. In: *International Conference on Electrical, Electronics, Communication Computer Technologies and Optimization Techniques, ICEECCOT 2017*,

vol. 2018-Janua, pp. 512–517. Institute of Electrical and Electronics Engineers Inc. ISBN 9781538623619.

Colgate, J.E., Wannasuphoprasit, W. and Peshkin, M.A. (1996). Cobots: robots for collaboration with human operators. In: *American Society of Mechanical Engineers, Dynamic Systems and Control Division (Publication) DSC*, vol. 58, pp. 433–439.

Dewangan, R.K., Shukla, A. and Godfrey, W.W. (2017 oct). Survey on prioritized multi robot path planning. In: *2017 IEEE International Conference on Smart Technologies and Management for Computing, Communication, Controls, Energy and Materials, ICSTM 2017 - Proceedings*, pp. 423–428. Institute of Electrical and Electronics Engineers Inc. ISBN 9781509059058.

Donald, B.R., Nagel, H.H., Bobrow, D.G. and Donald, B.R. (1987). A Search Algorithm for Motion Planning with Six Degrees of Freedom* Recommended by. Tech. Rep..

Encoder (2020). Document Directory > Encoder Products.
Available at: http://encoder.com/literature/optical-encoder-guide.pdf/

Faigl, J. (2017). Robotic Paradigms and Control Architectures B4M36UIR-Artificial Intelligence in Robotics. Tech. Rep..

Fan, T., Long, P., Liu, W. and Pan, J. (2020 jun). Distributed multi-robot collision avoidance via deep reinforcement learning for navigation in complex scenarios. *The International Journal of Robotics Research*, vol. 39, no. 7, pp. 856–892. ISSN 0278-3649.

Fiorini, P. and Shiller, Z. (1998 jul). Motion Planning in Dynamic Environments Using Velocity Obstacles. *The International Journal of Robotics Research*, vol. 17, no. 7, pp. 760–772. ISSN 0278-3649.

Garrido-Jurado, S., Muñoz-Salinas, R., Madrid-Cuevas, F.J. and Medina-Carnicer, R. (2016 mar). Generation of fiducial marker dictionaries using Mixed Integer Linear Programming. *Pattern Recognition*, vol. 51, pp. 481–491. ISSN 00313203.

Gasparetto, A. and Scalera, L. (2019). From the unimate to the delta robot: The early decades of industrial robotics. In: *History of Mechanism and Machine Science*, vol. 37, pp. 284–295. Springer.

Godoy, J., Karamouzas, I., Guy, S.J. and Gini, M. (2016). C-Nav: Implicit Coordination in Crowded Multi-Agent Navigation. Tech. Rep..

Hart, P.E., Nilsson, N.J. and Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107. ISSN 21682887.

Holte, R.C., Perez, M.B., Zimmer, R.M. and Macdonald, A.J. (1998). Hierarchical A*: Searching Abstraction Hierarchies Efficiently. Tech. Rep..

Hönig, W., Kumar, T.K.S., Cohen, L., Ma, H., Xu, H., Ayanian, N. and Koenig, S. (2016). Multi-Agent Path Finding with Kinematic Constraints. *undefined*.

Hrbáček, J., Ripel, T. and Krejsa, J. (2010). Ackermann mobile robot chassis with independent rear wheel drives. In: *Proceedings of EPE-PEMC 2010 - 14th International Power Electronics and Motion Control Conference*. ISBN 9781424478545.

Jenie, Y.I., van Kampen, E.J., de Visser, C.C. and Chu, Q.P. (2014). Velocity obstacle method for non-cooperative autonomous collision avoidance system for UAVs. In: *AIAA Guidance, Navigation, and Control Conference*. ISBN 9781600869624.

Jung, B. and Sukhatme, G.S. (2007 jun). Cooperative Multi-robot Target Tracking. In: *Distributed Autonomous Robotic Systems 7*, pp. 81–90. Springer Japan.

Kalman, R.E. (1960). Contributions to the Theory of Optimal Control. Tech. Rep..

Kant, K. and Zucker, S. (1985 jan). Trajectory Planning In Time-Varying Environments, 1: TPP = PPP + VPP. In: Casasent, D.P. and Hall, E.L. (eds.), *Intelligent Robots and Computer Vision*, vol. 0521, p. 220. SPIE.

Karaman, S. and Frazzoli, E. (2011 may). Incremental sampling-based algorithms for optimal motion planning. In: *Robotics: Science and Systems*, vol. 6, pp. 267–274. ISBN 9780262516815. ISSN 2330765X. 1005.0416.

Kavathekar, P. and Chen, Y. (2011). Vehicle platooning: A brief survey and categorization. In: *Proceedings of the ASME Design Engineering Technical Conference*, vol. 3, pp. 829–845. ISBN 9780791854808.

Kavraki, L.E., Švestka, P., Latombe, J.C. and Overmars, M.H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580. ISSN 1042296X.

Kline, R.R. (2011 oct). Cybernetics, automata studies, and the dartmouth conference on artificial intelligence. *IEEE Annals of the History of Computing*, vol. 33, no. 4, pp. 5–16. ISSN 10586180.

Koenig, N. and Howard, A. (2004). Design and use paradigms for Gazebo, an open-source multi-robot simulator. In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 3, pp. 2149–2154. ISBN 0780384636.

Koenig, S. and Likhachev, M. (2005). Fast Replanning for Navigation in Unknown Terrain. *IEEE TRANSACTIONS ON ROBOTICS*, vol. 21, no. 3.

Koenig, S., Likhachev, M. and Furcy, D. (2005). Lifelong Planning A*. Tech. Rep..

Konolige, K., Guzzoni, D. and Nicewarner, K. (2002). A Multi-Agent System for Multi-Robot Mapping and Exploration. In: *Multi-Robot Systems: From Swarms to Intelligent Automata*, pp. 11–19. Springer Netherlands.

Korf, R.E. (1990 mar). Real-time heuristic search. *Artificial Intelligence*, vol. 42, no. 2-3, pp. 189–211. ISSN 00043702.

Kuipers, B., Feigenbaum, E.A., Hart, P.E. and Nilsson, N.J. (2017 mar). Shakey: From conception to history.

Laumond, J.-P. (1987). Finding collision-free smooth trajectories for a non-holonomic mobile robot. Tech. Rep..

Laumond, J.P., Sekhavat, S. and Lamiraux, F. (2006 jan). Guidelines in nonholonomic motion planning for mobile robots. In: *Robot Motion Planning and Control*, pp. 1–53. Springer-Verlag.

LaValle, S.M. (1998). Rapidly-Exploring Random Trees: A New Tool for Path Planning. vol. 129, pp. 98–11. ISSN 1098-6596. arXiv:1011.1669v3.

LaValle, S.M. (2006 jan). *Planning algorithms*, vol. 9780521862. Cambridge University Press. ISBN 9780511546877.

Lerman, K., Jones, C., Galstyan, A. and Mataríc, M.J. (2006). Analysis of dynamic task allocation in multi-robot systems. *International Journal of Robotics Research*, vol. 25, no. 3, pp. 225–241. ISSN 02783649.

Li, B. and Zhang, Y. (2019). Real-Time Trajectory Planning with Static and Dynamic Obstacles for an Automated Guided Vehicle. , no. February.

Likhachev, M., Ferguson, D., Gordon, G., Stentz, A. and Thrun, S. (2005). Anytime Dynamic A*: An Anytime, Replanning Algorithm. Tech. Rep..

Likhachev, M., Gordon, G. and Thrun, S. (2004). ARA*: Anytime A* with Provable Bounds on Sub-Optimality. Tech. Rep..

Long, P., Fanl, T., Liao, X., Liu, W., Zhang, H. and Pan, J. (2018 sep). Towards optimally decentralized multi-robot collision avoidance via deep reinforcement learning. In: *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 6252–6259. Institute of Electrical and Electronics Engineers Inc. ISBN 9781538630815. ISSN 10504729. 1709.10082.

Masehian, E. and Sedighizadeh, D. (2010 aug). Multi-objective robot motion planning using a particle swarm optimization model. *Journal of Zhejiang University: Science C*, vol. 11, no. 8, pp. 607–619. ISSN 18691951.

Maseko, B.B., Van Daalen, C.E. and Treunicht, M.J. (2020). Optimised Path Planning and Path Tracking for Autonomous Vehicles with Constrained Kinematics in ROS. Tech. Rep..

McKinsey & Company (2019). Industrial robotics: Insights into the sector's future growth dynamics. Tech. Rep..

Mellah, S., Graton, G., El Adel, E.M., Ouladsine, M. and Planchais, A. (2018 dec). On fault detection and isolation applied on unicycle mobile robot sensors and actuators. In: *2018 7th International Conference on Systems and Control, ICSC 2018*, pp. 148–153. Institute of Electrical and Electronics Engineers Inc. ISBN 9781538685372.

Micro Robotics (2020). Stepper motor.
Available at: https://www.robotics.org.za/35BYG312P1?search=steppermotornema

Mohd Salih, J.E., Rizon, M. and Yaacob, S. (2006 may). Designing Omni-Directional Mobile Robot with Mecanum Wheel. *American Journal of Applied Sciences*, vol. 3, no. 5, pp. 1831–1835. ISSN 15469239.

Moor, J. (2006 dec). The Dartmouth College Artificial Intelligence Conference: The Next Fifty Years. *AI Magazine*, vol. 27, no. 4, pp. 87–87. ISSN 2371-9621.

Mountz, M., D'Andrea, R. and Wurman, P.R. (2006). Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. Tech. Rep. 1.

Nascimento, T.P., Dórea, C.E. and Gonçalves, L.M.G. (2018 may). Nonholonomic mobile robots' trajectory tracking model predictive control: A survey. *Robotica*, vol. 36, no. 5, pp. 676–696. ISSN 14698668.

Ni, Z., Wang, T. and Liu, D. (2015 jul). Survey on medical robotics. *Jixie Gongcheng Xuebao/Journal of Mechanical Engineering*, vol. 51, no. 13, pp. 45–52. ISSN 05776686.

Parker, L.E. and Head, A. (2010). Multi-Robot Path Planning and Motion Coordination. Tech. Rep..

PBC Linear (2020). Stepper Motor Datasheet. Tech. Rep..
Available at: https://resi.store/datasheets/nema17.pdf

Pendleton, S.D., Andersen, H., Du, X., Shen, X., Meghjani, M., Eng, Y.H., Rus, D. and Ang, M.H. (2017 mar). Perception, planning, control, and coordination for autonomous vehicles. *Machines*, vol. 5, no. 1, p. 6. ISSN 20751702.

Qin, S.J. and Badgwell, T.A. (2003 jul). A survey of industrial model predictive control technology. *Control Engineering Practice*, vol. 11, no. 7, pp. 733–764. ISSN 09670661.

Quigley, M. (2009). ROS: an open-source Robot Operating System. *ICRA Workshop on Open Source Software*, vol. 3, no. 2, pp. 5–11.

Quigley, M., Berger, E. and Ng, A.Y. (2007). STAIR: Hardware and Software Architecture. Tech. Rep..

RasPi.TV (2016). How Much Power Does Raspberry Pi3B Use?
Available at: https://raspi.tv

RobotHallOfFame (2020). The Robot Hall of Fame - Powered by Carnegie Mellon University.
Available at: http://www.robothalloffame.org/inductees/03inductees/unimate.html

Romero-Ramirez, F.J., Muñoz-Salinas, R. and Medina-Carnicer, R. (2018 aug). Speeded up detection of squared fiducial markers. *Image and Vision Computing*, vol. 76, pp. 38–47. ISSN 02628856.

ROS (2020*a*). move_base - ROS Wiki.
Available at: http://wiki.ros.org/move{_}base

ROS (2020*b*). ROS.org — About ROS.
Available at: https://www.ros.org/about-ros/

Siegwart, R. and Nourbakhsh, I. (2004). Introduction to Autonomous Mobile Robotos. pp. 48–67.

Silver, D. (2005). Cooperative Pathfinding. Tech. Rep..
Available at: www.aaai.org

Snape, J., Van Den Berg, J., Guy, S.J. and Manocha, D. (2010). Smooth and collision-free navigation for multiple robots under differential-drive constraints. In: *IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems, IROS 2010 - Conference Proceedings*, pp. 4584–4589. ISBN 9781424466757.

Solomonoff, R.J. (1985 jan). The time scale of artificial intelligence: Reflections on social effects. *Human Systems Management*, vol. 5, no. 2, pp. 149–153. ISSN 01672533.

Stentz, A. (1994). Optimal and Efficient Path Planning for Partially-Known Environments. Tech. Rep..

Stentz, A. (1995). The Focussed D* Algorithm for Real-Time Replanning. Tech. Rep..

Texas Instruments (2020). LM2596 datasheet. Tech. Rep..
Available at: https://www.ti.com/lit/ds/symlink/lm2596.pdf?ts=1601997508159

Van Berg, J.D., Lin, M. and Manocha, D. (2008). Reciprocal velocity obstacles for real-time multi-agent navigation. In: *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 1928–1935. ISBN 9781424416479. ISSN 10504729.

Van Den Berg, J., Guy, S.J., Lin, M. and Manocha, D. (2011*a*). Reciprocal n-body collision avoidance. In: *Springer Tracts in Advanced Robotics*, vol. 70, pp. 3–19. Springer, Berlin, Heidelberg. ISBN 9783642194566. ISSN 16107438.

Van Den Berg, J., Snape, J., Guy, S.J. and Manocha, D. (2011*b*). Reciprocal collision avoidance with acceleration-velocity obstacles. In: *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 3475–3482. ISBN 9781612843865. ISSN 10504729.

Viljoen, R. (2020). Decentralized cooperative navigation for autonomous ground vehicles (AGV) - YouTube.
Available at: https://www.youtube.com/watch?v=nmoqPvkSkOY{&}feature=youtu.be

Whitaker, N. (2006). Darpa urban challenge.
Available at: https://www.darpa.mil/about-us/timeline/darpa-urban-challenge

Wilkerson, J.L., Bobinchak, J., Culp, M., Clark, J., Halpin-Chan, T., Estabridis, K. and Hewer, G. (2014). Two-Dimensional Distributed Velocity Collision Avoidance. Tech. Rep..

Wit, J.S. (2000). Vector pursuit path tracking for autonomous ground vehicles. Tech. Rep..

Wooden, D.T. (2006). Graph-based Path Planning for Mobile Robots. Tech. Rep..

Xu, X. (2020). PathFinding.js.
Available at: https://qiao.github.io/PathFinding.js/visual/

Yan, Z., Jouandeau, N. and Cherif, A.A. (2013 dec). A Survey and Analysis of Multi-Robot Coordination. *International Journal of Advanced Robotic Systems*, vol. 10, no. 12, p. 399. ISSN 1729-8814.

Appendix A

Bill of Materials

Table A.1: Bill of materials used for each vehicle.

Part name	**Part code**	**Quantity**
Stepper motors	42BYGHW208	2
Chassis	Custom	1
Wheel	FSR90R-W	2
Stepper breakout board	SEP-MOD	2
Stepper driver	6970622931867	2
Voltage regulator	LM2596-DIS-MOD	1
Battery management system	3S 20A 18650 charger	1
Li-Ion battery	18650	3
Battery holder	18650-WIRE-3X	1
Switch	MTS-103	2
Ribbon cables	RIB-COMBO-20	6
Raspberry Pi 3B	RP3B	1

Appendix B

Cooperative Trajectory Planning Examples

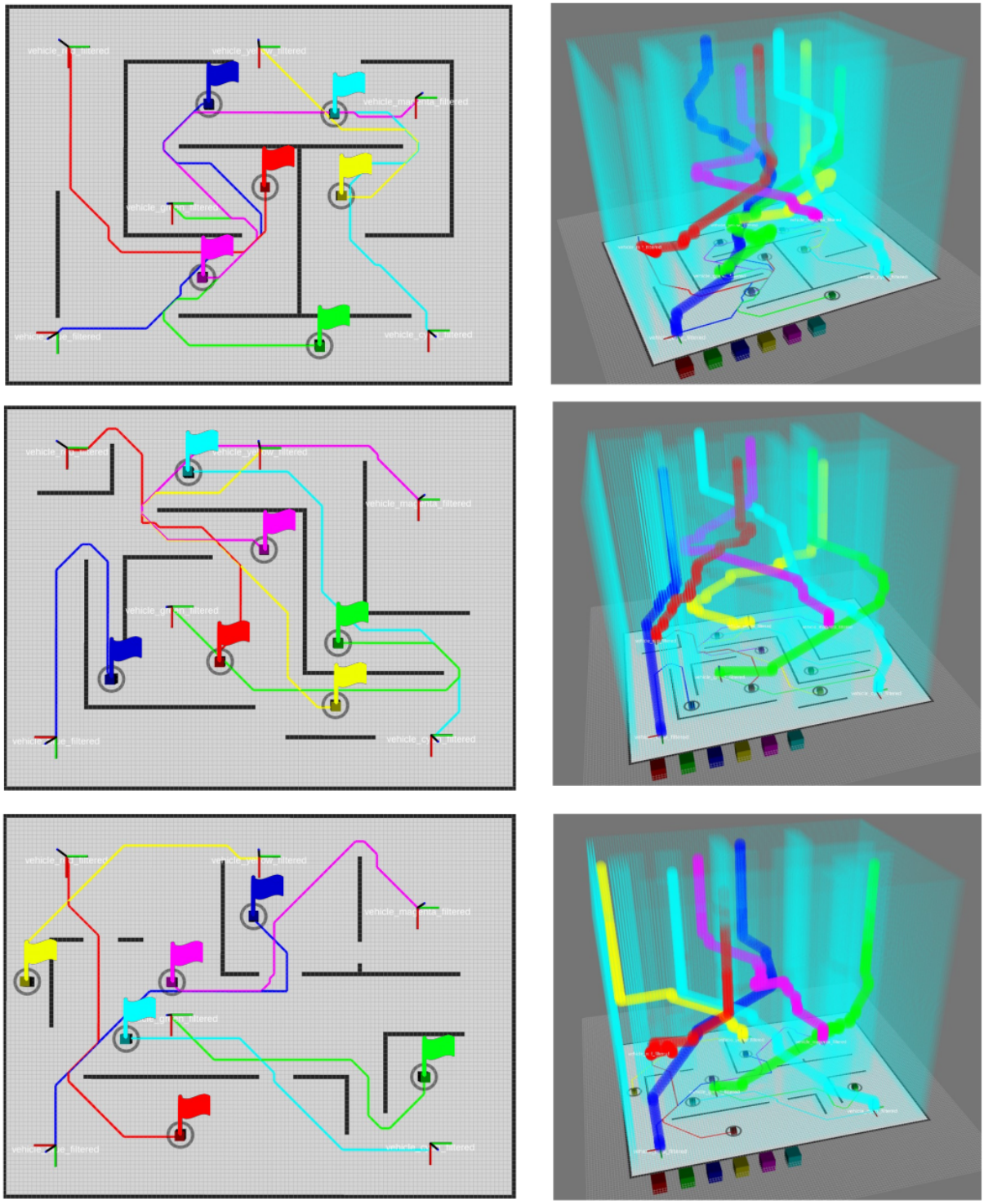


Figure B.1: Additional examples of trajectories produced by the cooperative trajectory planner.

Appendix C

Box drive results

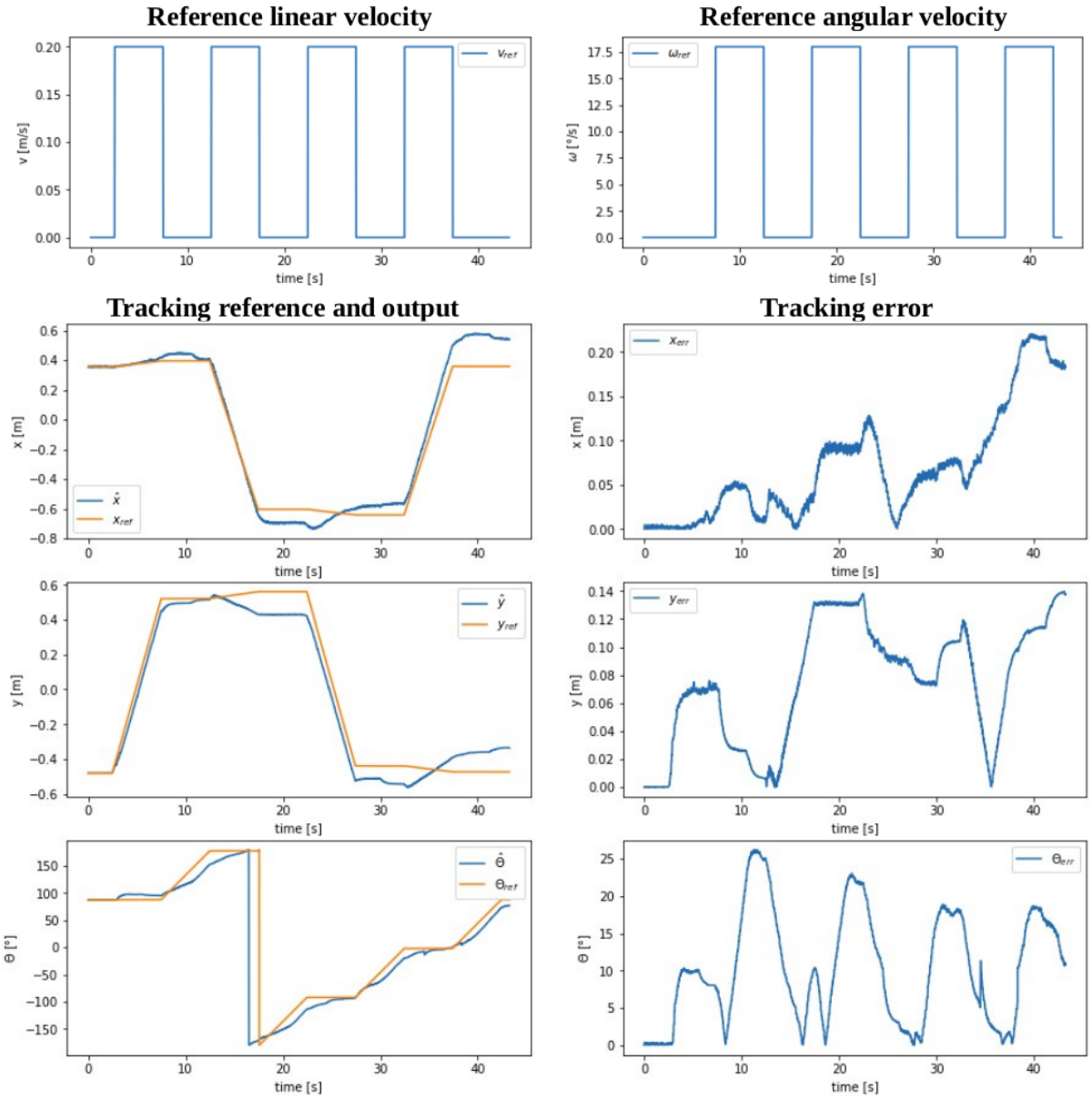


Figure C.1: The time series plots recorded during vehicle one's box drive test in the anticlockwise direction.

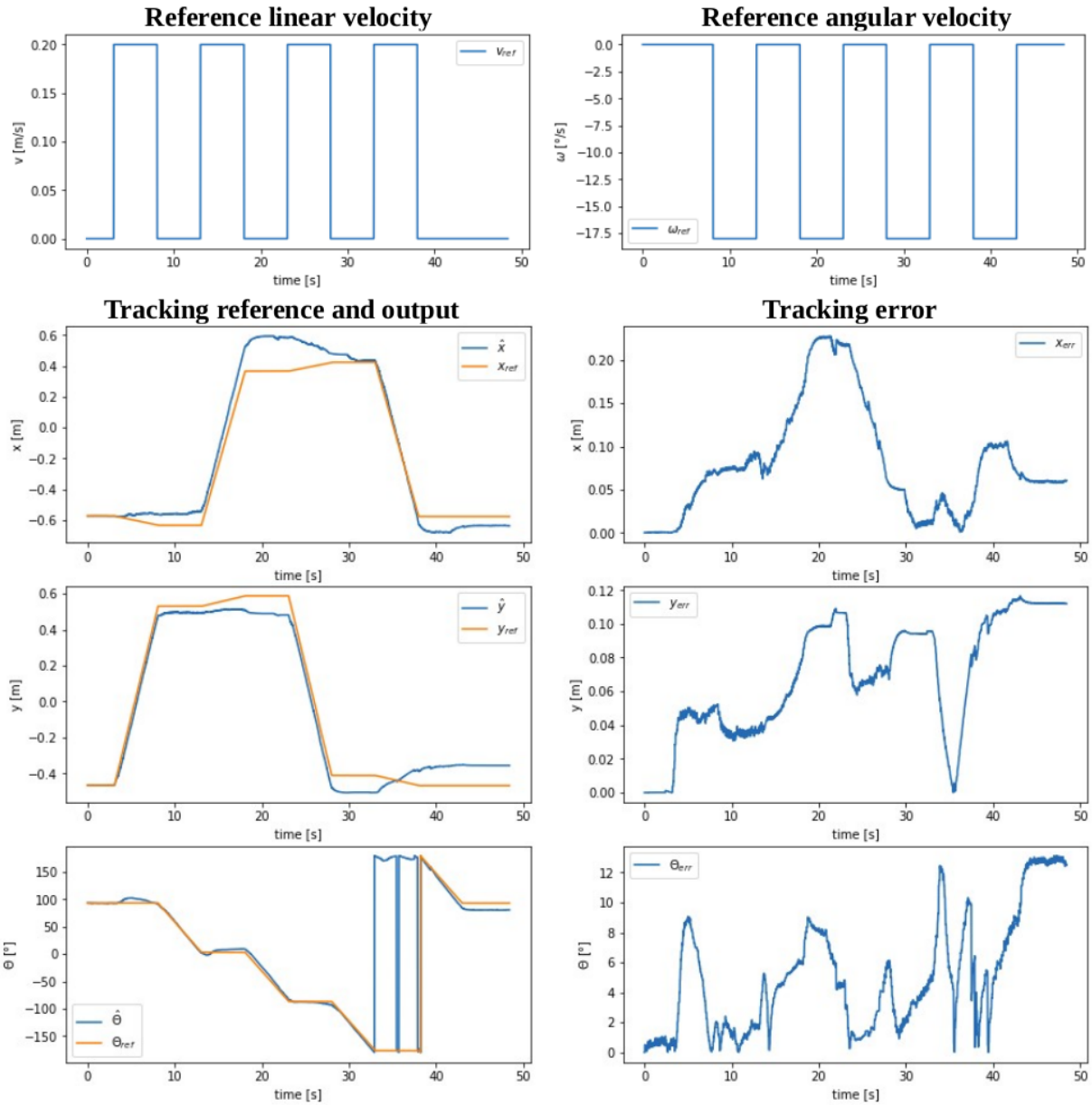


Figure C.2: The time series plots recorded during vehicle two's box drive test in the clockwise direction.

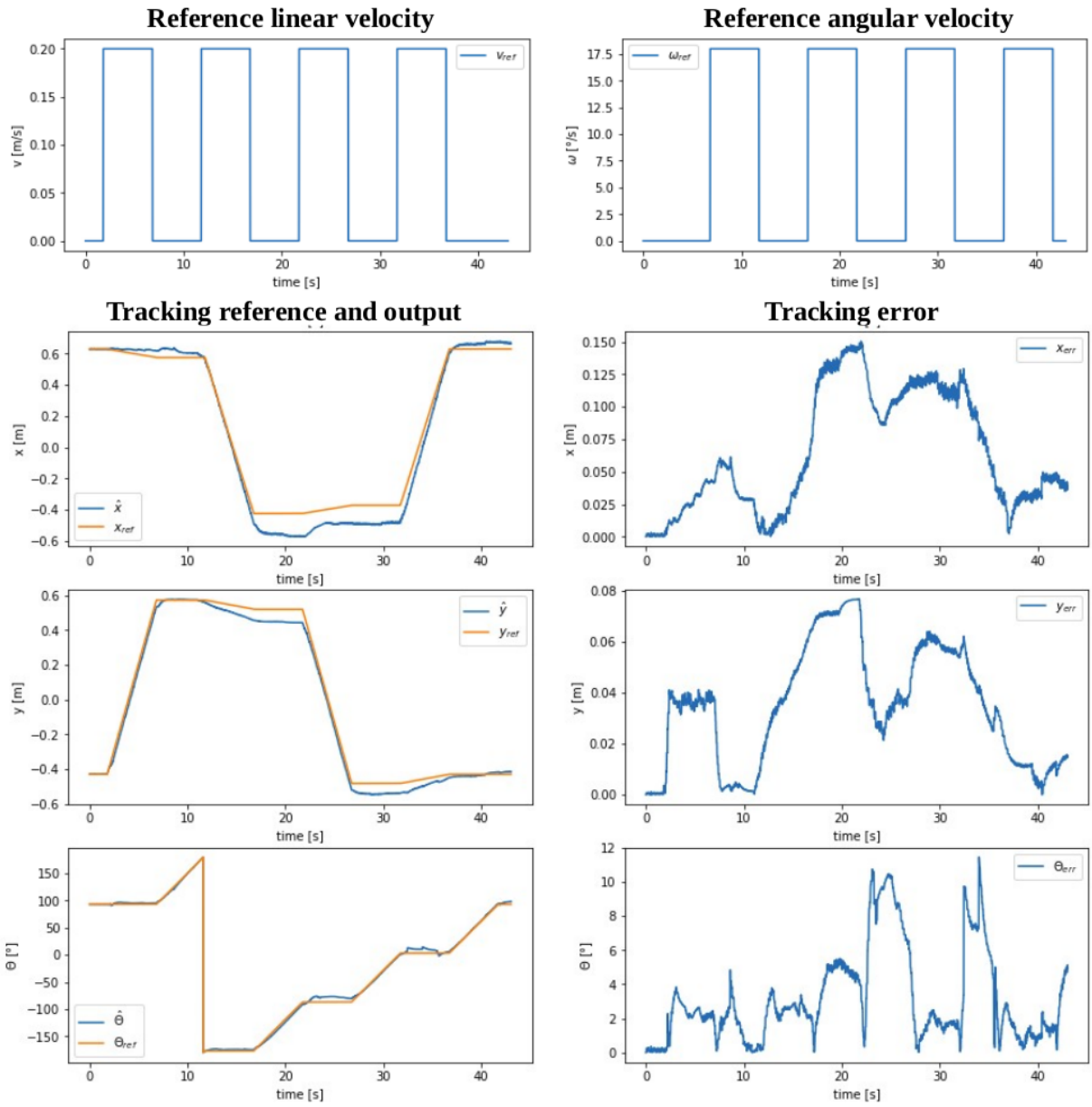


Figure C.3: The time series plots recorded during vehicle two's box drive test in the anticlockwise direction.

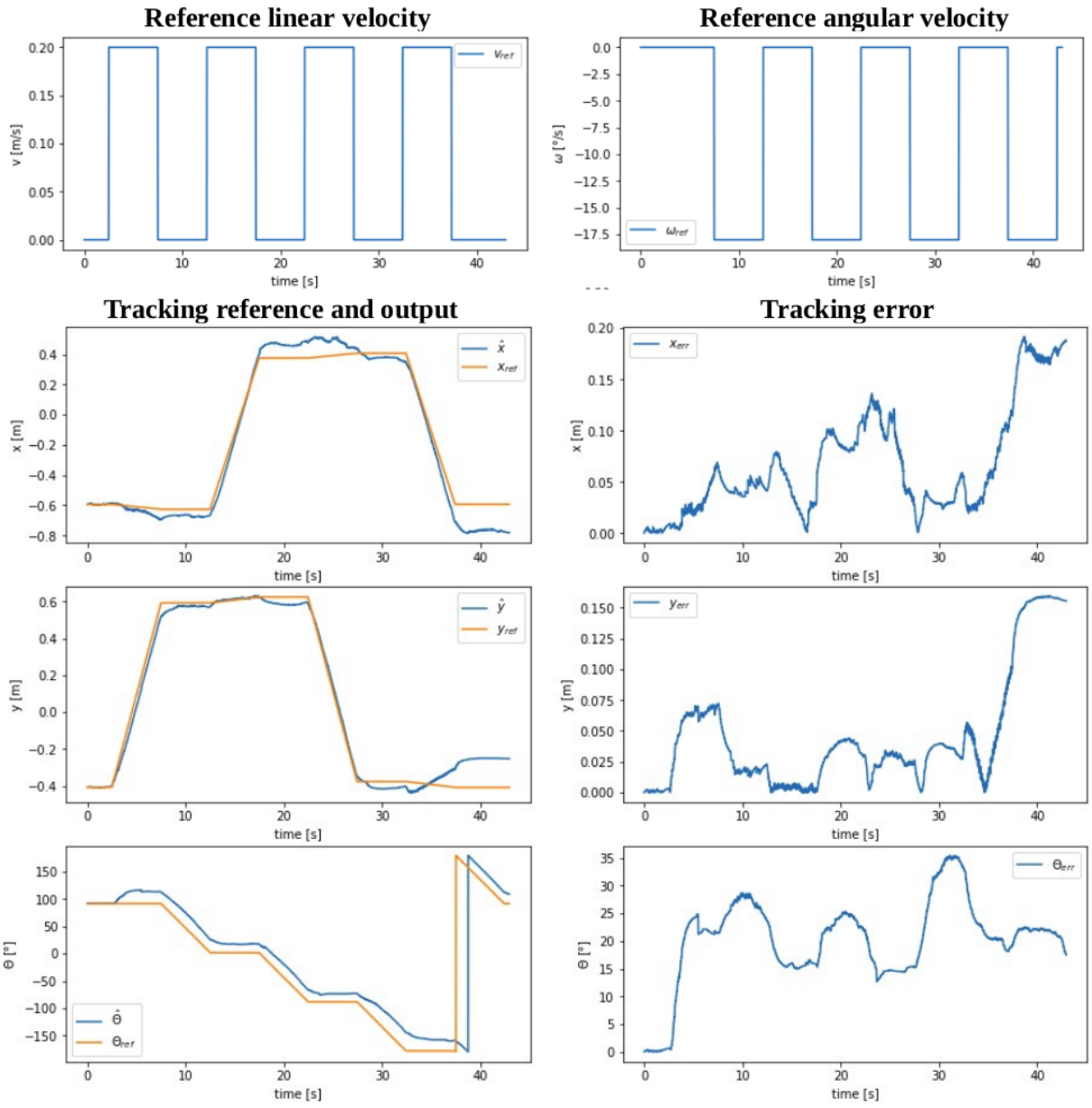


Figure C.4: The time series plots recorded during vehicle three's box drive test in the clockwise direction.

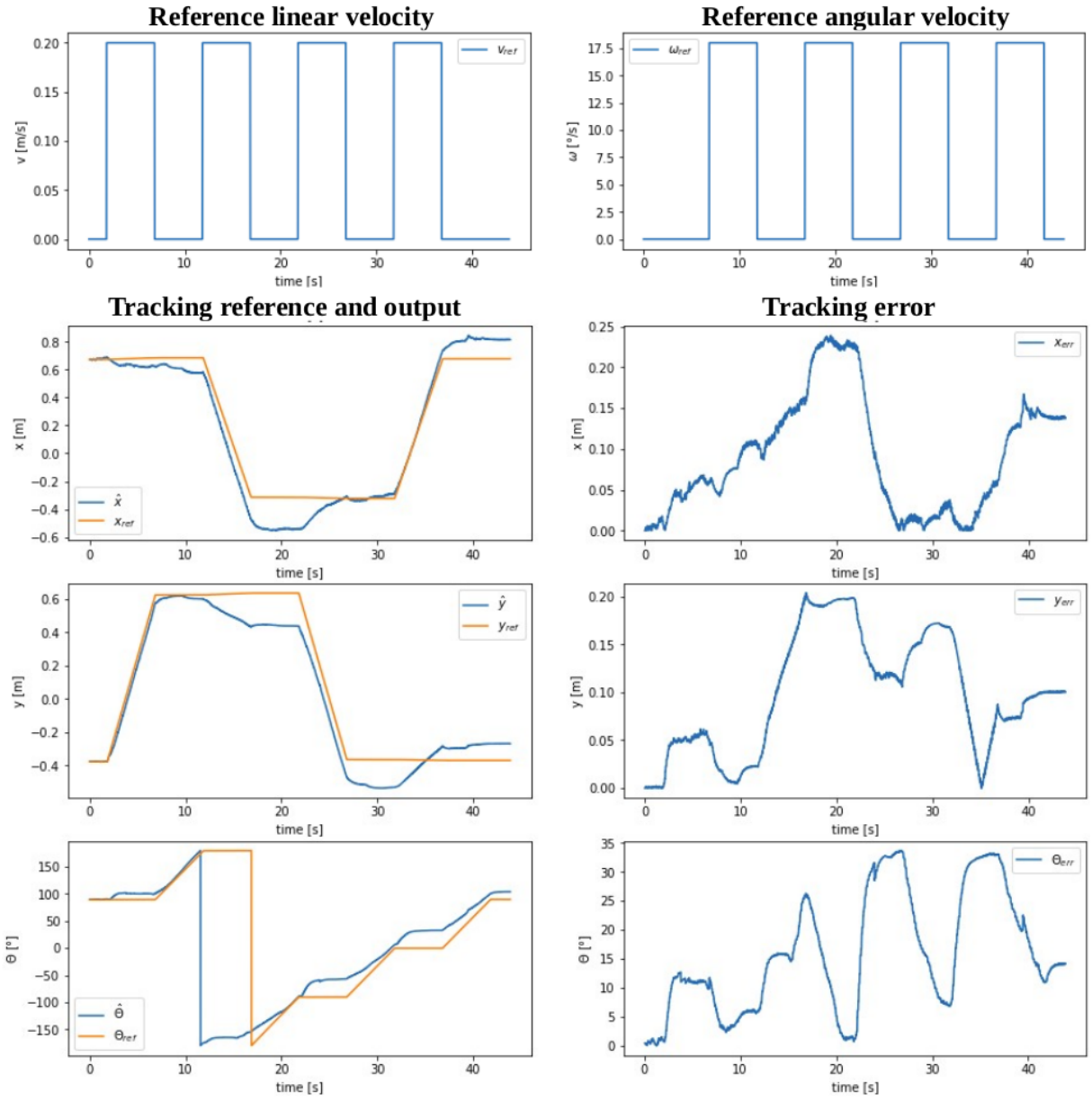


Figure C.5: The time series plots recorded during vehicle three's box drive test in the anticlockwise direction.

Appendix D

Trajectory Deviation Graphs

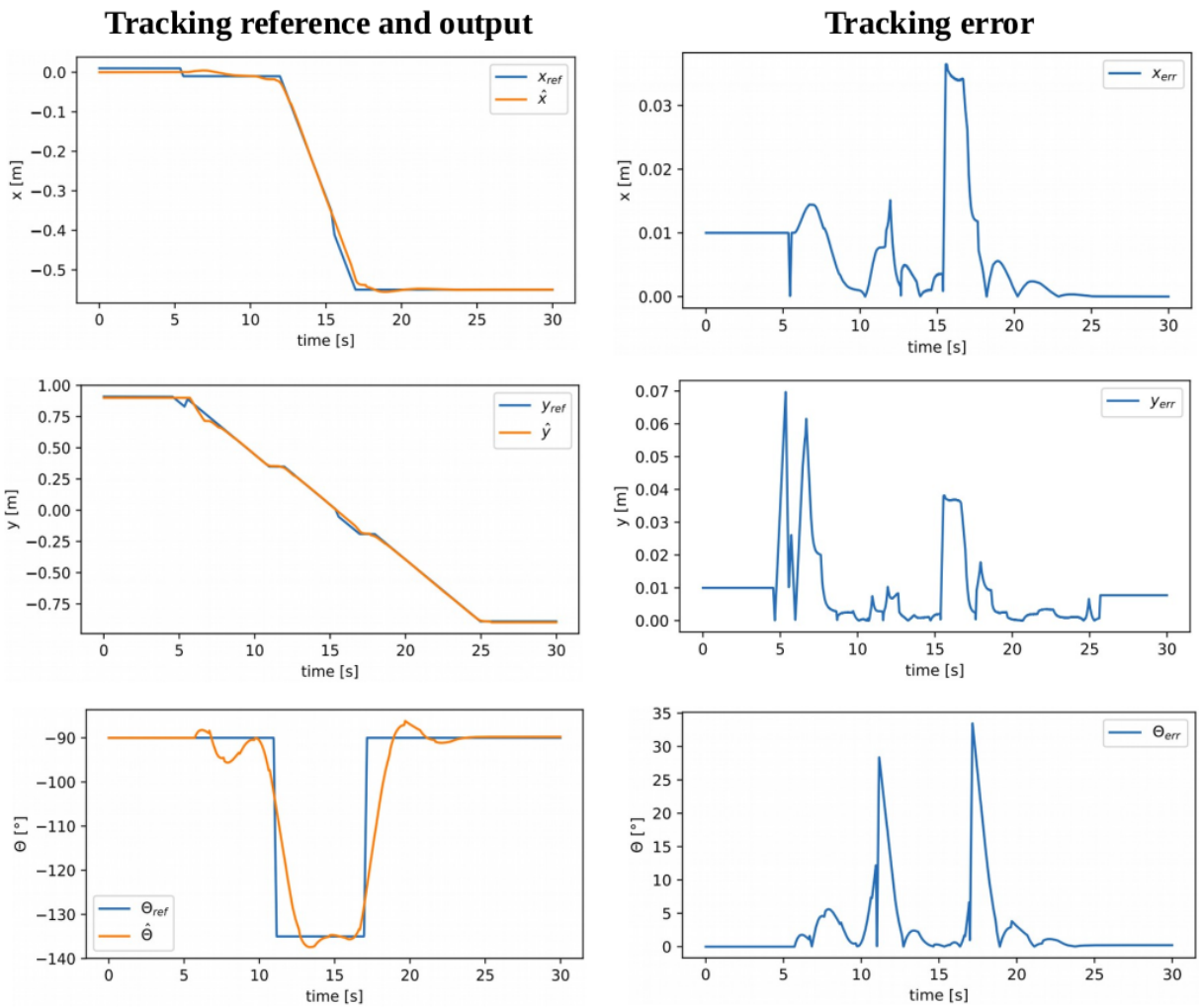


Figure D.1: The plots of the reference and estimated values of the LiteSim green vehicle's state over time, as well as the error.

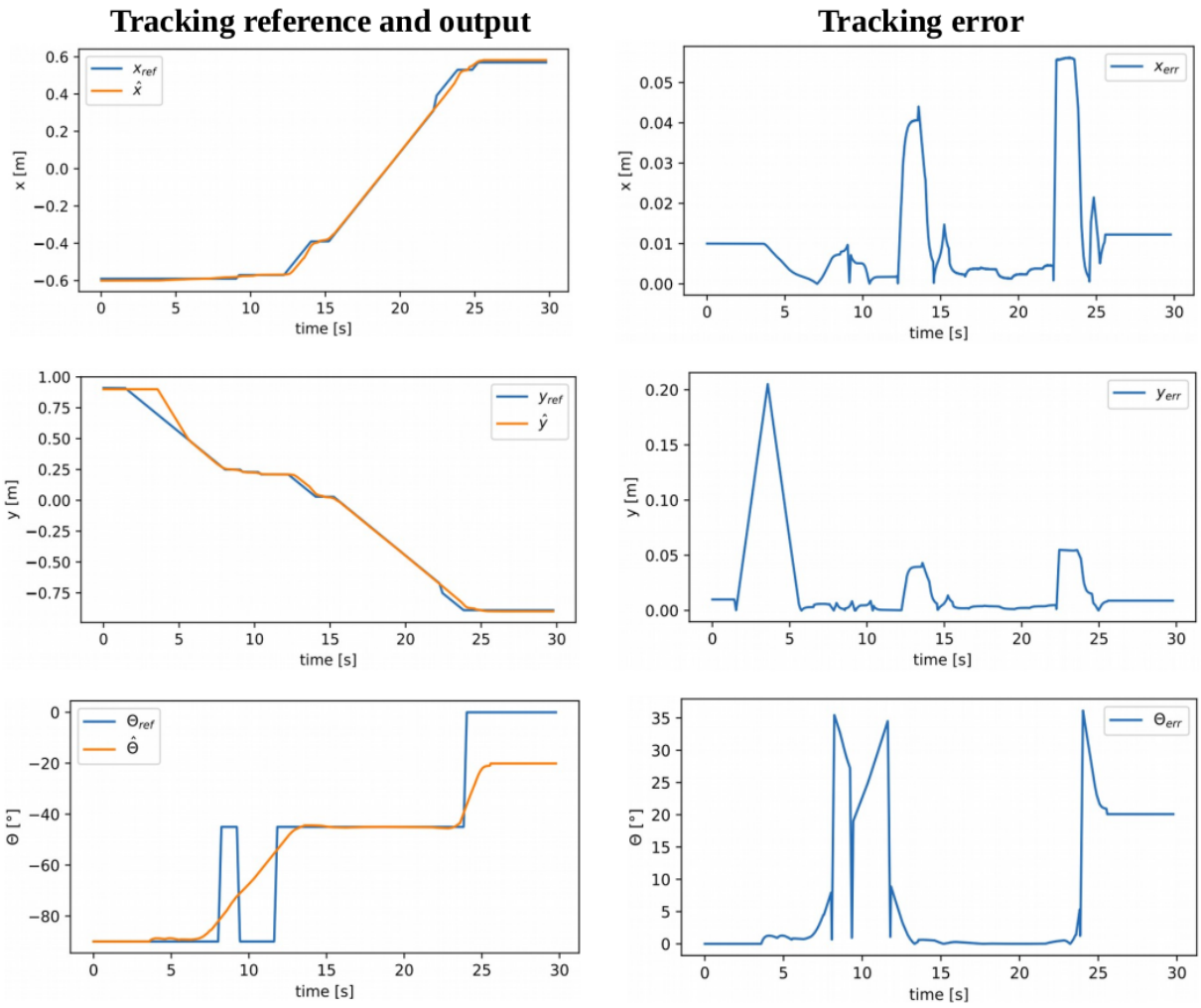


Figure D.2: The plots of the reference and estimated values of the LiteSim blue vehicle's state over time, as well as the error.

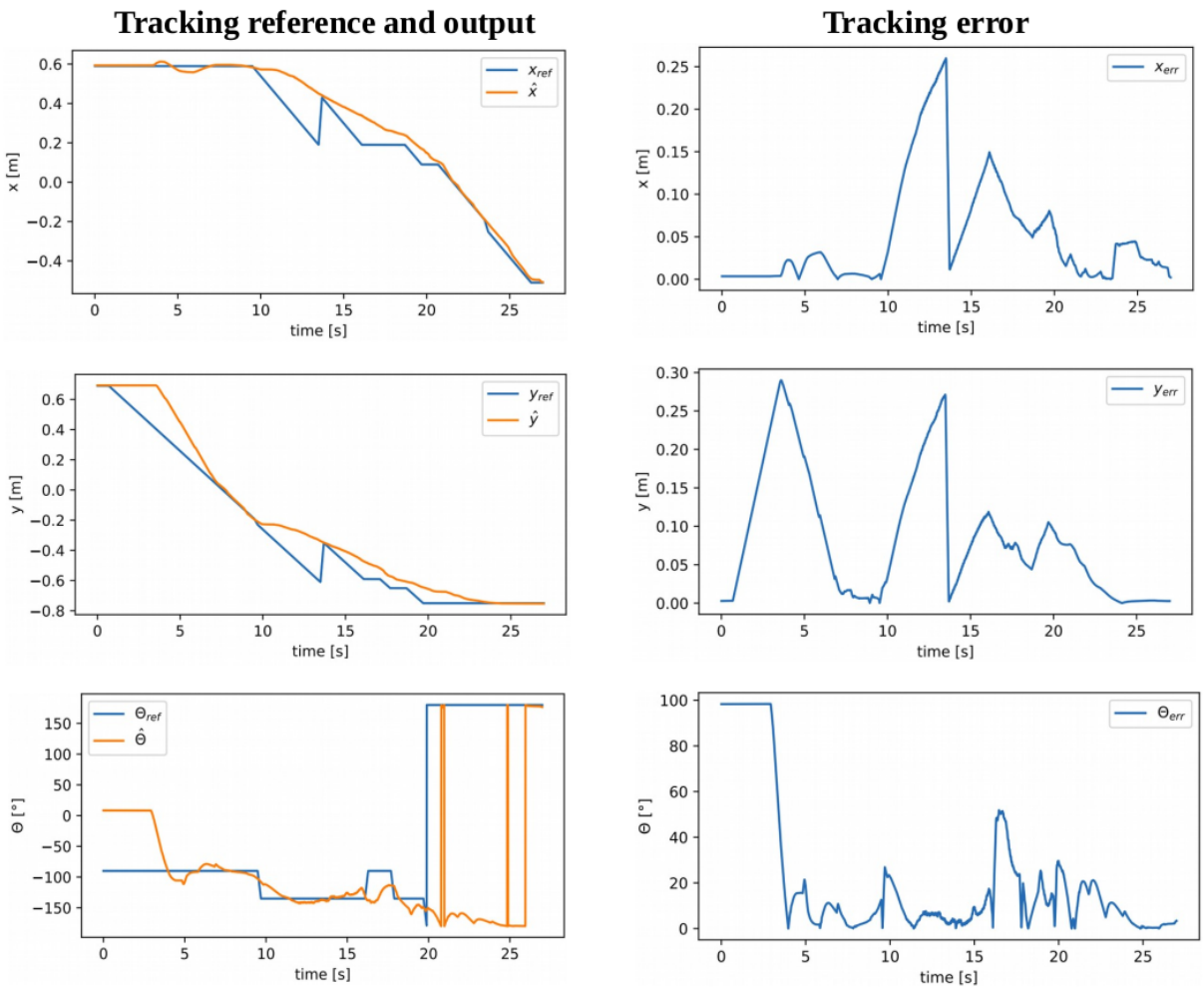


Figure D.3: The plots of the reference and estimated values of the GazeboSim green vehicle's state over time, as well as the error.

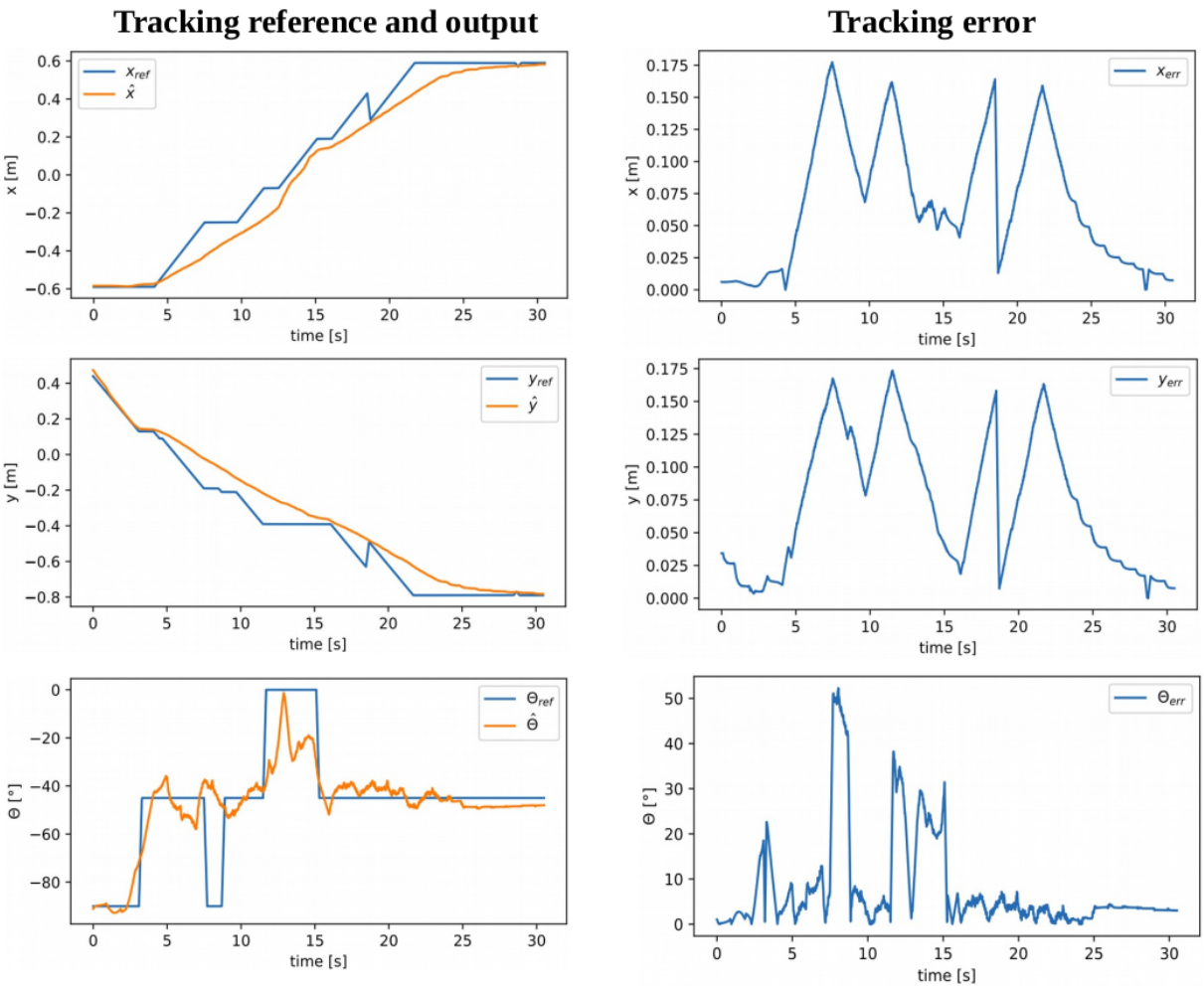


Figure D.4: The plots of the reference and estimated values of the GazeboSim blue vehicle's state over time, as well as the error.

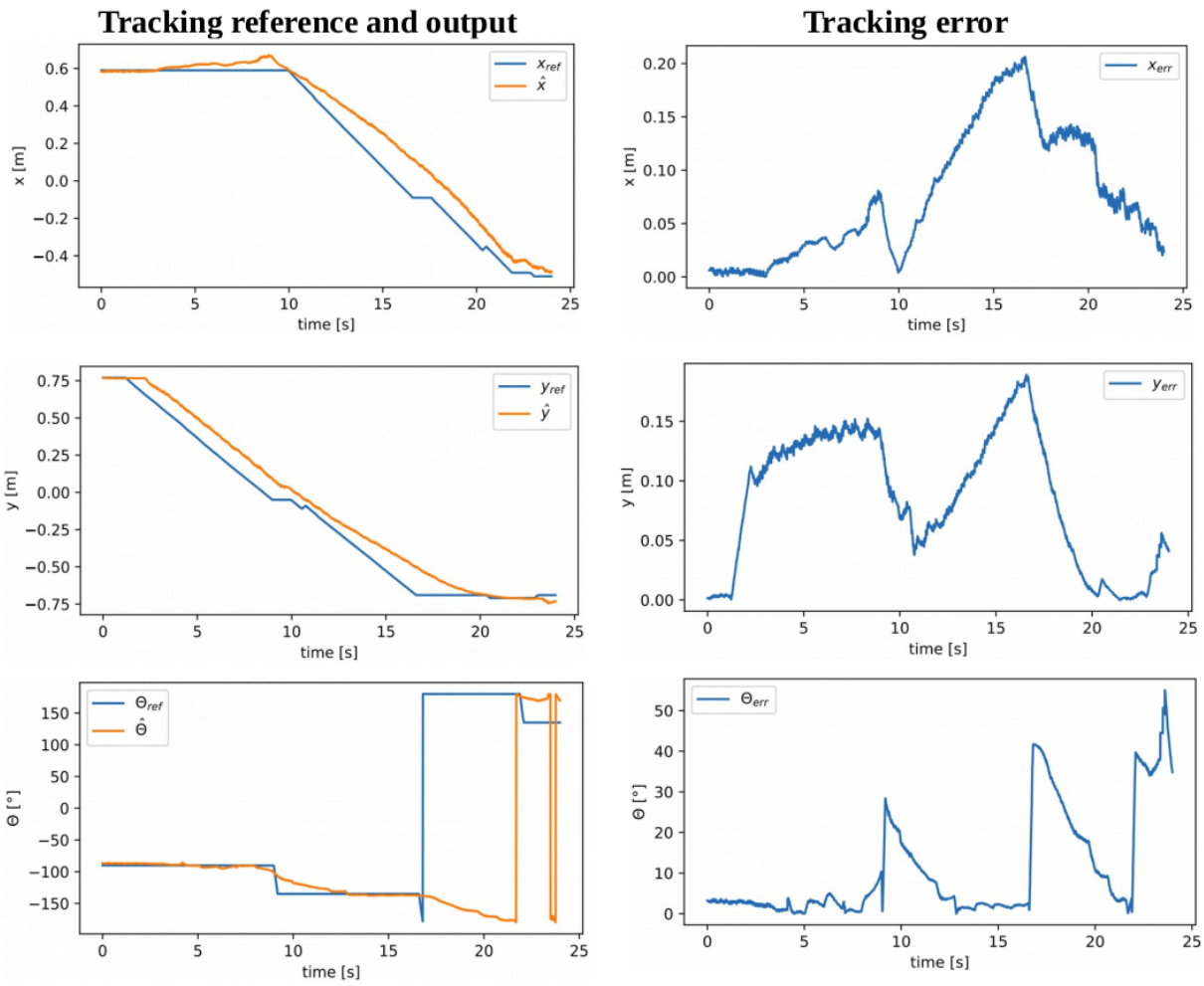


Figure D.5: The plots of the reference and estimated values of the practical test green vehicle's state over time, as well as the error.

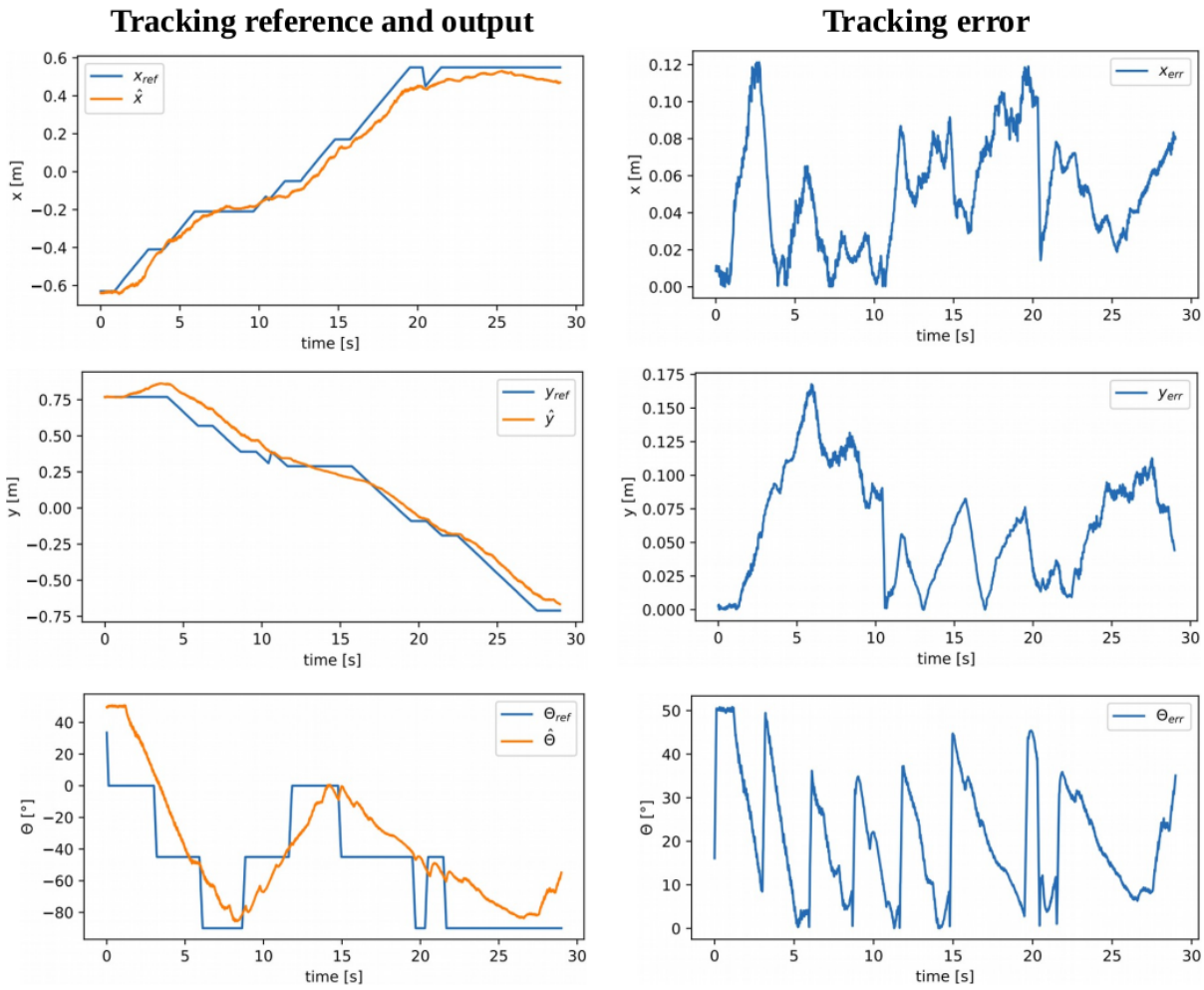


Figure D.6: The plots of the reference and estimated values of the practical test blue vehicle's state over time, as well as the error.